8-20-2015 12:00 AM

Generating Invalid Input Strings for Software Testing

Benjamin D. Revington
The University of Western Ontario

Supervisor
James H. Andrews
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

Recommended Citation

Revington, Benjamin D., "Generating Invalid Input Strings for Software Testing" (2015). *Electronic Thesis and Dissertation Repository*. 3087.
https://ir.lib.uwo.ca/etd/3087

GENERATING INVALID INPUT STRINGS FOR SOFTWARE TESTING

(Thesis Format: Monograph)

by

Benjamin Daniel Revington

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario
August, 2015

Abstract

Grammar-based testing has interested the academic community for decades, but little work has been done with regards to testing with invalid input strings. For our research, we generated LR parse tables from grammars. We then generated valid and invalid strings based on coverage of these tables. We evaluated the effectiveness of these strings in terms of code coverage and fault detection by inputting them to subject programs which accept input based on the grammars. For a baseline, we then compared the effectiveness of these strings to a more general approach where the tokens making up each string are chosen randomly. Analysis revealed that the LR strategy achieved higher code coverage than the randomly generated strings in most cases. Even when the LR strategy garnered less code coverage, it still covered substantial code not touched by the benchmark. The LR strategy also showed effective fault finding ability for some programs.

Keywords: Software Testing, LR Parse Table, Grammar-Based Testing, Invalid Input Testing, Code Coverage

Acknowledgments

First and foremost, I owe my gratitude to Dr. Jamie Andrews. He provided guidance, technical expertise, the choice of topics, help with the algorithms, and so much more. This thesis would not have been possible without him. I am truly blessed to have had such an excellent supervisor. I could not have asked for a more intelligent, kind, or patient man to serve under.

I thank Drs. Mike Bauer, David Bellhouse, and Sylvia Osborn for examining my thesis and coming to my defense. Additionally, I wish to acknowledge Dr. Duncan Murdoch. His instruction on the statistical programming language R years ago was what interested me in computer science initially.

I wish to thank Janice Wiersma and the rest of the department for making my Masters at Western possible.

I will always be indebted to my family for their love and support. I would not be where I am today without their help.

I am thankful for the support from all my friends. I wish to specifically mention Santo Carino for providing some technical expertise, and Megan Stuckey, for her kindness, inspiration, and offering me usage of her family's resources when needed.

I wish to acknowledge the creators of CUP for creating a publicly available LR parsing program that provided exactly what we needed to perform this research.

Finally, I thank my Lord and Saviour, Jesus Christ. Without Him, I am nothing.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Software testing is an important part of the software development life cycle, the process of creating or modifying software systems. Software testing is the stage in the software development cycle where the software's quality is evaluated with the aim of detecting problems so that they can be fixed before the program is put into production. Grammar-based testing is the branch of software testing aimed at fixing problems in software which require their inputs to conform to a particular grammar. Significant research has been done on grammar-based testing in evaluating *valid* inputs that conform to a grammar, but limited work has been done in evaluating *invalid* inputs. An invalid input will likely test different code than a valid input. In this thesis, we explore generating both valid and invalid inputs for grammar-based testing purposes. We use a structure called an LR parse table to generate valid and invalid inputs from grammars, and then feed the inputs to software programs expecting inputs conforming to these grammars. We measure the effectiveness of this procedure by observing the code coverage and analyzing the outputs of the programs.

1.2 Motivations

Software testing is an increasingly important area of research as it is one of the most costly stages of the software development life cycle [9]. Improperly tested software could potentially result in millions of dollars in losses, or in some cases, even injury or death. Extensive research has been done on testing programs dealing with simple integer inputs, but programs accepting grammar-based string inputs have typically been more difficult to test [5]. Since many grammar-based software programs exist, it is necessary to ensure that grammar-based testing procedures are adequate.

Work in the field of grammar-based testing has traditionally focussed on generating valid inputs whilst neglecting invalid inputs. It is not sufficient for a software program to simply accept and handle valid inputs correctly; it needs to gracefully reject invalid inputs as well. Programmers will add code to prevent software from crashing if it receives invalid input, preferably providing the user with helpful error messages.

One example of invalid input would be if the user did not finish entering all the necessary elements of their input. For example, a person writing a C program might accidentally leave off a "}" at the end of their code. The compiler would need to catch and inform the programmer of the error without crashing. Another possibility would be if the user makes a typo, such as accidentally writing a ":" when a ";" is required. In this situation, the input might be complete, but one element of the input has been exchanged for an invalid one.

Evaluating these situations by measuring the code coverage achieved by detecting invalid inputs, and analyzing the outputs to ensure that the programs tested were rejecting erroneous output correctly, would have great value in the field of grammar-based testing. We evaluate both of these situations in this thesis.

1.3 Thesis Focus

The focus of this thesis is to analyze the effects of invalid inputs in grammar-based testing. Our approach was to generate strings from LR parse tables, convert them to bytes acceptable as inputs, and submit these inputs to grammar-based software. The results were evaluated by observing the code coverage achieved and checking if the software under test classified the inputs correctly.

An LR parse table is an unchanging data table generated by an LR parser. LR parsers are a type of bottom-up parser that efficiently handles deterministic context-free languages in linear time [1]. We generated these LR parse tables from context-free grammars. LR parse tables have a collection of states which are reachable with different strings. We searched through the LR parse tables by generating strings that take us to different states. We used a heuristic search algorithm to find routes through the LR parse table. In this manner, we generated three types of strings: strings that were complete and valid, strings that were complete but invalid because one valid token in the string had been exchanged with an invalid one, and strings that were invalid because they were incomplete. We attempted to generate as many strings in each category as we could, but due to the presence of infinite routes through the LR parse tables, strings going through some states were infeasible.

We also generated strings not based on the LR parse table to use as a comparison. These strings had their tokens chosen randomly, with no thought given to the structure of the LR parse table. Randomized testing has been shown to be surprisingly effective in practice, finding problems even in well-tested software programs [5]. As such, we felt that this randomized testing approach would make for a fair evaluation of our LR parse table inspired strategies for generating strings.

After generating these strings, they were converted into bytes suitable for input to a software program. We obtained software programs that take input corresponding to the grammars from which we generated strings. We then provided our inputs to the software programs for the testing procedure.

We calculated the amount of code covered in the software programs tested, in order to evaluate how thorough our approach was at stimulating the grammar-based software. Our results indicate that the LR parse table approach usually covered more code than the randomly generated inputs, and when it did not, it still held complementary value. We also evaluated the outputs of each program to determine whether the inputs had been correctly discerned as valid or invalid. We discovered that the number of correctly classified inputs varied greatly between subject programs. Additionally, we observed if any inputs caused the subject programs to crash, to determine if our strategy could detect any lurking bugs in these well-tested programs. We discovered that multiple inputs caused crashes.

1.4 Thesis Organization

The introduction, motivation for our work, and focus of this thesis, have been provided in Chapter 1. Chapter 2 discusses related work to provide context for this thesis. We also provide an in-depth explanation of some important concepts for our research. We discuss an example grammar to illustrate some key concepts in Chapter 3. In Chapter 3, we also define key terms, and describe the algorithms used in this thesis. Chapter 4 is where we introduce our subject grammars, describe how we prepared them, explain how we generated parse tables, detail how we tuned the parameters of our heuristic search algorithm, and discuss how we generated our inputs. In Chapter 5, we identify our subject programs, describe how the programs were prepared for testing, give the procedure used in applying the inputs to the programs, and provide an analysis of the experimental results. In Chapter 6, we conclude and discuss possible areas of future work.

Chapter 2

Background and Related Work

This chapter details some related work and provides some background knowledge for the reader. We discuss the nature of automated software testing, explain grammars and how to derive LR parse tables from them, and then provide a brief history of the study of generation of grammar-based input.

2.1 Automated Software Testing

Software programs can be tested by black box or white box testing. Black box testing determines test cases based on the requirements for the program on a high level, while white box testing determines test cases based on investigating the code itself [5]. This thesis combines the two approaches, since the test cases are generated based on a high level view of the grammar that the program under test looks at, but the results are measured partly based on the amount of code covered.

The purpose of testing software programs is to locate faults in code. A software developer programming a software application may accidentally make an error that causes an unintended result in the code. A fault is a manifestation of an error in software that may cause a failure when running the code. A fault causes the program to behave in an unintended manner, such as crashing or providing a user with incorrect output. Finding and localizing faults in programs, and the effects that they have on the software under test, is an ongoing area of research in software engineering. Francis et al. [15] use a tree based method for analyzing the faults of programs and determining why they cause tests to fail. Masri et al. [24] performed a study of the causes of reduced effectiveness in localizing faults of programs. In our research, we examine finding faults themselves, but we also observe what coverage may result in programs when input that has the opportunity of finding a fault is provided.

2.1.1 Test Coverage

Test coverage, also known as code coverage, refers to the extent to which the source code of a software program has been tested by a given collection of tests. If a program's test suite achieves more code coverage, then it means that it has been tested more thoroughly, and so it is likely that more bugs in the program may have been caught. In this thesis, we use line coverage, and, to a lesser extent, branch coverage, to assess the amount of code coverage for our four subject programs. Line coverage checks to see how many lines of a program have been covered. Branch coverage tests if each decision direction in the program has been executed.

As an example, consider the following code snippet:

```
if (x > 5 && y > 5) { x = x+1;
y = x*2;
 x = x - 9; }
```

If we run only one test case, and in it x and y are less than 5, then we will have 33% line coverage because the top line was executed but not the other two. We would also have 50% branch coverage, because one of the two possible truth values for the if statement have been tested (namely, the false value). If we add another test case where both x and y are greater than 5, then we will have achieved 100% line coverage since all lines are now covered (between both test cases). This would also give us 100% branch coverage because we have evaluated both options (true and false) for the if statement. It is worth differentiating line coverage from the slightly stronger metric of statement coverage, which checks if each statement of the program has been tested. Since x = x +1 is on the same line as the if statement above, the statement coverage value will be different from line coverage if the if statement is false for all test cases.

Branch coverage differs from condition coverage in that condition coverage checks if each Boolean sub-expression evaluates to both true and false but does not worry about the overall decision made. In the example above, condition coverage would check if both x >5 evaluates to true and false, and if y >5 evaluates to both as well, but condition coverage is not concerned about whether the whole decision (what is between the brackets) evaluates to true or false. We look at both branch and line coverage in this thesis, but place the focus on line coverage, because our code coverage measuring utilities differ slightly in how they define branch coverage.

The test coverage utilities that we used for this thesis are Cobertura and gcov. Cobertura is a free Java tool that calculates the amount of code covered by tests. We used Cobertura version 2.1.1 to test Java subject programs. gcov is a coverage tool that comes with the free C compiler gcc. We used gcov version 1.01 on a subject program written in C to obtain line coverage and gcov's version of branch coverage.

2.1.2 Randomized Testing

In running our code, we wished to see if inputs following the structure of a grammatical structure known as an LR parse table (which will be explained in the next subsection) would prove more helpful, less helpful, redundant to, or complementary to, randomized inputs. Testing with random inputs, or randomized testing as it is commonly called, is unbiased and has shown to be surprisingly effective [26] [16] [5], so it makes for a worthy comparison. Randomized testing hammers the program with random input, which often catches unexpected software bugs that more structured approaches miss. Miller et al. [26] used randomized testing on Unix utilities in this manner and were able to find many bugs in well-tested programs. However, they admit that their approach is not exhaustive, as that would require formal verification. Groce et al. [16] later used random testing on aircraft software as a means for preparing the way for using formal verification. They achieve this through their approach to randomized test input generation, hardware simulation with fault injection, automated test case minimization, and designing the initial system with an eye for testability later.

Randomized testing has a long history, but has questions as to its level of thoroughness due to its undirected nature. Pacheco et al. [28] attempted to deal with this difficulty with feedback-directed random test generation. As test inputs are executed, feedback is analyzed and used for improving further tests. This approach maintains the benefits of random testing with regards to scalability and simplicity while avoiding some of its downsides, such as meaningless or redundant inputs. Korel [22] describes a technique for choosing the inputs while the program is still in execution. This dynamic strategy of automatically generating test data makes use of how information flows through a program to hasten the search process for inputs that cause undesired behaviour in the program [25] [22].

Andrews et al. [3] used a genetic algorithm to optimize test coverage by finding parameters for randomized unit testing. Unit testing programs allows for testing of individual sections of code, rather than taking a high level approach based on system requirements [13]. Andrews et al. [3] generated unit test data on two levels: a randomized unit testing engine lower level, and a genetic algorithm that uses selection, mutations, fitness evaluation, and recombination to find good values for the randomized unit testing parameters.

In contrast to randomized testing, symbolic testing executes the path of a program and generates a set of constraints on the input variables [21] [10] [8]. This approach, however, suffers from the state space explosion problem, as many different paths are possible through a program. Our approach was based on a context-free grammar and then applied concretely to a program rather than being built directly on the program under test. Symbolic testing is not practical for normal usage due to the state space explosion problem, but it does locate program faults effectively.

2.2 Grammars

2.2.1 Context-Free Grammars

A grammar gives a clearly defined syntactic specification of a language. From some types of grammars, we can establish a syntactic structure for a source program by creating a parser. A parser receives a string of tokens from a lexical analyzer and verifies that the string can be produced for the source language by the grammar. A properly designed grammar's parser can be valuable for such tasks as generating correct object code from source programs, and for detecting errors in input. The four main types of errors that can be detected are lexical errors (misspellings of keywords, identifiers, or operators), syntactic errors (syntax errors such as a misplaced semicolon or a missing "{"), semantic errors (such as type mismatches between operands and operators), and logical errors (incorrect reasoning by the programmer that may still be a well-formed construction of the language) [1]. Evaluating grammars is important for software testing since many programs expect some inputs to conform to a grammar. According to Hennessy and Power [17], "Grammar-based software includes not only programming language compilers, but also tools for program analysis, reverse engineering, software metrics and documentation generation."

A context-free grammar is made up of a start symbol, terminals, non-terminals, and productions. Terminals are the basic symbols from which strings are formed. Non-terminals denote patterns of tokens, and enforce a hierarchical structure on the language. One non-terminal is declared as the start symbol, and it denotes the set of strings making up the language produced by the grammar. The mode in which non-terminals and terminals form strings is specified by the productions for the context-free grammar. Each production is denoted by a non-terminal on the left side, a “—>” or “::=” (based on preference, as the meaning is the same for both symbols), and then a body on the right side made up of zero or more non-terminals or terminals that describe the way the strings of the non-terminal at the left side may be constructed. An example production looks like this:

S ::= a S T b

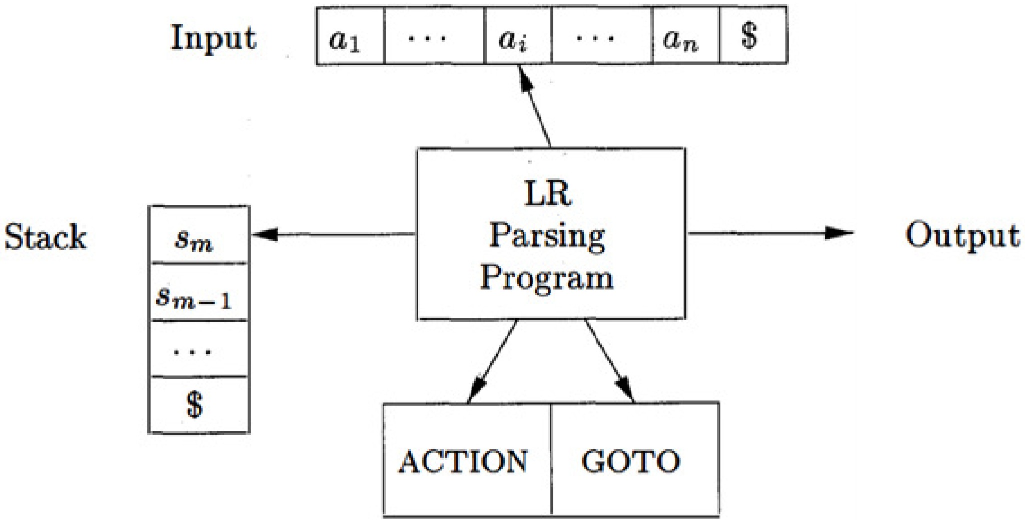
where “S” is the non-terminal on the left side, “a” and “b” are terminals, and “S” and “T” are non-terminals [1].

2.2.2 LR Parsers

A bottom-up parser is one that constructs a parse tree (a graphical representation that filters out the production order to replace the non-terminals for a derivation of the grammar) for an input string by starting at the bottom (the leaves of the tree), and works its way up to the top or root of the tree. The most common kind of bottom-up parser used is the LR(k) parser. The “L” refers to scanning the input from left to right. The “R” refers to constructing a rightmost derivation in reverse. The “k” is the number of symbols of look ahead used in making parsing decisions. In practical applications, k is 0 or 1. LR parsers use constructs known as LR parse tables, which can be created to recognize almost all programming language constructs that can be generated from context-free grammars [1].

A schematic for an LR parser is given in Figure 2.1. The LR parser consists of a stack of states, an input list of tokens read one at a time, a driver program, a parse table

Figure 2.1 LR Parser Schematic [1].



consisting of two parts ("action" and "goto") and the output [1].

The behaviour of an LR parser follows a set pattern, with the current input token, a_i, and state on the top of the stack, s_m, determining the resulting configuration. The consequences of parsing one token with a particular stack of states are determined by the action-goto table. There are four basic options for the parse of a given input token:

1. The parser shifts the next state, s, as described by the action-goto table, onto the state stack. The token a_i is read and the parser moves on to token a_{i+1}.
2. The parser performs a reduction based on a particular production in the grammar, where r state symbols are popped off the stack leaving state s_{m-r} at the top. The value of r is determined by the number of tokens on the right hand

side of the production. After popping r states off the stack, the parser pushes a new state on the stack based on the non-terminal on the left side of the production and the state currently exposed at the top of the stack after the r states have been popped. The goto part of the action-goto table determines what the new top state will be. A reduction does not use up an input token, so the configuration of the parser after will still be looking at token a_i, just with a different state on the top of the stack.

3. If the parser reaches a final state and receives the proper token, the LR parser accepts the input and the parsing is completed.

4. The parser may find an error, such as if a particular input token is not accepted at a particular state [1].

A parser's next action is determined by the LR parse table. We will now look at an example from [1] for illustration. An augmented grammar (augmented means that the start production ending with EOF has been added) is shown in Figure 2.2. The LR parse table derived from the grammar is shown in Figure 2.3. Figure 2.3 shows a $ in place of EOF, either of which indicates a termination of the input. Blanks in the LR parse table indicate that there are no transitions at those states for those tokens (i.e. an error will result). In Figure 2.3, transitions starting with s are shifts and transitions beginning with r are reductions. The numbers given by the shifts indicate the state to shift to. The numbers given by the reductions indicate the number of the productions to reduce with (where S ::= CC is production 1). For the given example, reaching the end of input at state 1 indicates that the input is accepted (which is equivalent to reducing with the start production).

The three main types of LR parser are the simple LR parser, the canonical LR parser,

Figure 2.2 Example grammar for Figure 2.3 from [1].

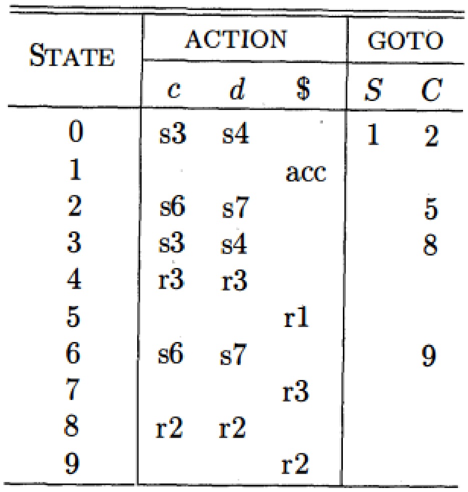
```
$START ::= S EOF

S ::= C C

C ::= c C

C ::= d
```

Figure 2.3 Sample Canonical LR Parse Table for Example Grammar [1].



STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

and the LALR parser. The simple LR parser (or LR(0) parser) is the easiest to create and is space efficient but has no look ahead. The most general form of LR parser is the canonical LR parser, also known as an LR(1) parser (LR(1) as parsers rarely require a k >1). The canonical LR parser is far less space efficient, but can accept more options due to the look ahead. In this thesis, we utilized an LALR or Look-Ahead LR parser. LALR parsers are nearly as powerful as canonical LR parsers but have the same number of states as the LR(0) parsers [1].

We decided to use a pre-existing, publicly available LALR parser generator, the Constructor of Useful Parsers (CUP) version 0.10 created by Scott E. Hudson [19]. CUP is similar to the well known compiler-compiler YACC in terms of role, features, specifications, etc. Unlike YACC however, "CUP is written in Java, uses specifications, including embedded Java code, and produces parsers which are implemented in Java" [19]. CUP also requires some additional information for an input specification beyond just the basic grammar, but this will be explored more later in this thesis.

There were some other automatic LR parser generators that we looked at, the best one being the GOLD Parsing System [11]. GOLD was simpler in setup than CUP, provided better documentation, and claimed to support more programming languages than any other parser. Unfortunately, GOLD did not provide us with as direct access to the LR parse table itself, which is what we needed.

2.3 Generation of Grammar-Based Input

There has been a significant amount of work done on generating grammar-based input over the years. In 1972, Purdom [29] described an algorithm that could be used to generate sentences for testing parsers. The goal of the algorithm is to produce short sentences from a context free grammar such that every production is used at least once. The sentences aid in debugging context free grammars and testing parsing programs. The algorithm aims for creating many short sentences rather than fewer, longer sentences. The parses are sent to a referee routine and the sentences to a parsing routine, with the parsing routine sending any parses it finds to the referee routine for double checking. Purdom reports that his method is fast, general, and complete enough to be valuable for detecting errors in a parsing building program, with the sentences generated having largely been useful in detecting errors in grammars.

Our research focussed more on the coverage of the individual states of the LR parse tables, while Purdom focussed on coverage of the grammar rules used to create the table. Our work also included generating invalid input in addition to valid input. While Purdom's valid input could aid in ensuring that the grammar did not reject good input, our work allows us to check that the grammar does not accept bad input too.

Zhao and Lyu [32] do further work on generating grammar-based input. They demonstrate the first approach to generating string test data automatically from character string predicates. Character string predicates are made up of one string comparison function, like `strcmp`, and one character string variable [32]. Our approach in this thesis is not to give programs input based on paths through the program directly, but rather to automatically generate our string test data by finding paths through the LR parse tables.

Hennessy and Power [17] explore the idea of generating minimal test suites for grammar-based software by a reduction strategy based on “rule coverage”, which analyses the rules (productions) that the grammar is based on. Minimizing a test suite ensures that the test suite is more efficient and will use fewer resources to execute. Not running unnecessary tests or parts of tests helps save software businesses money [31] [9]. The approach Hennessy and Power use was tested on three grammar-based tools for C++. Hennessy and Power report that their test suite is comparable to that generated by Purdom's algorithm in terms of size. Additionally, Hennessy and Power's approach is more semantically correct. Hennessy and Power also found that while there was no strong correspondence between rule coverage and code coverage for a grammar-based software application, their reduced test suites kept the code coverage reasonably stable. Unfortunately, their minimized test suites failed to preserve the fault-finding ability of the originals.

When we generated string test data through various techniques, we found that the state space for these strings could be massive. This is a common issue with grammar-based testing. For a particular grammar "one can trivially derive combinatorially exhaustive test-data sets up to a specified depth" [23]. Lämmel and Schulte [23] attempt to control for this issue of state space explosion in grammar-based testing. They use a bottom-up algorithm for test data generation based on a collection of control mechanisms that are simpler approximations of full combinatorial coverage. An example of something that they did to control for state space explosion is to control recursive calls, both in terms of total depth and by keeping the depth of recursions with multiple arguments balanced. They implement their approach in a C# based test data generator called Geno. Their approach is effective, but would not work for our research since we derive the strings from parse table paths rather than program paths.

Another approach to generating test data is to use genetic algorithms [18] [2]. Alshraideh and Bottaci [2] present an approach to test data generation where program branches are covered which depend on string predicates. They use genetic algorithms to search through the programs, and string matching techniques like regex, string ordering, and string equality, to generate string test data.

In industry, a lot of testing needs to be done on programs accepting strings as input. It is difficult to obtain strings for testing that achieve high code coverage and fault-finding ability. Beyene and Andrews [5] address this problem by deriving Java classes called “Grammatical Category Objects” (GCOs) from a given context-free grammar. They use meta-heuristic and deterministic techniques to obtain strings. Every symbol (terminal or non-terminal) receives its own GCO. They test valid strings, but did not test invalid strings as we do in this thesis. They test their approach on two example grammars and their corresponding subject programs just as we do in this thesis.

Kiezun et al. [20] created the HAMPI program to provide a solver over bounded string variables for string constraints. HAMPI is “a solver for word equations over strings, regular expressions, and context-free grammars” [20]. A user specifies constraints and HAMPI finds a string that satisfies all the constraints or reports that the constraints are unsatisfiable. String terms in the HAMPI language are created out of bounded string variables, string constants, string concatenations, and extraction operations. The smallest level of formula (i.e. atomic formulas) in the HAMPI language are “equality over string terms, the membership predicate for regular expressions and context-free grammars, and the substring predicate that takes two string terms and asserts that one is a substring of the other” [20]. Bounding of context-free grammars and regular expressions is an important feature of HAMPI that differentiates it from the string-constraint solvers used in analysis and testing tools. Bounding is beneficial because it permits a more expressive input language that allows operations on context-free languages that would otherwise be undecidable. Bounding also makes the satisfiability problem that HAMPI solves easier to deal with. Due to a multitude of dependencies on specific versions of specific libraries, we decided not to use HAMPI for this thesis.

Massé et al. [6] build off the work done by Kiezun et al. They mention that Kiezun et al. tried solving string constraints using regular expressions, but state that this approach cannot express morphisms (substitutions) and antimorphisms (reverse substitutions) well. The authors propose a new representation of fixed length word equations through the use of a graph data structure. They then implement a generic word equation solver in Python to assess the feasibility of their representation. Like Massé et al., Büttner and Cabot [7] emphasize the importance of modelling for problems involving strings [30]. They create a lightweight solver using constraint logic programming for tractable string constraints in model finding that creates large solutions. Their implementation solves some large instances of common string constraints efficiently.

Feldt and Poulding [14] attempt to generate string test data with meta-heuristic techniques, but in a different manner than what was attempted by Beyene and Andrews. Feldt and Poulding [14] use random generators and formal grammars to generate highly structured grammar-based inputs. They give the operator the option of biasing them towards scalability (a bias towards large structures), detecting faults efficiently, or predicting the reliability of the software in regular usage.

Chapter 3

Process and Algorithms

This chapter describes the process and algorithms used to obtain strings. We demonstrate the process with an example grammar provided by CUP. The chapter explains the example grammar, defines some terminology, demonstrates these definitions with examples, and discusses the algorithms used in searching through the LR parse tables.

3.1 Example From User Manual

In this portion of the thesis, we will discuss the workings of an LR parse table for a grammar using a simple example grammar provided with CUP. This grammar had no subject program for testing on, but it will help the reader to understand some basic concepts. While programming the algorithms to manipulate the larger grammars, this sample grammar was used for much of the early testing.

3.1.1 CUP Specification

The CUP specification syntax is given as five major components: package and import specifications, used code components, symbol (terminal and non-terminal) lists, precedence declarations, and the context-free grammar itself. All of the components must appear in the order given, though some are optional. In Figures 3.1 and 3.2, we give the specification for the example grammar that CUP provides. The example specification is for a simple calculator. This calculator permits using parentheses to clarify order of operations, provides access to the modulus operation, and requires lines to be terminated with a semicolon (;).

The first component of the CUP specification is package and import specifications for the generated Java code. This portion of the CUP specification is optional. In the example above and our own input grammars, we imported the `java_cup.runtime` classes since these provided helpful data for our program.

The next part of the grammar specification is an optional part for user code components (which in our example appears under the "Preliminaries" comment). These basic declarations aid in telling CUP how to set up and use the scanner (used for reading tokens). Like the import and package declarations, there was no need to add anything here for our purposes, so we left in the same declarations found in the example grammar.

Next is the first required portion of the CUP specification: the symbol lists. The symbol lists are declarations of all the terminals and non-terminals used in the grammar. CUP's output includes a Java class where each terminal (and optionally non-terminal) is identified by an integer. It is possible to specify class names for the symbols, such

Figure 3.1 CUP specification imports, preliminaries, terminals, non-terminals, and precedences for an example grammar.

```
// CUP specification for a simple expression evaluator (no actions)

import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal            SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal            UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;

/* Non terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr, term, factor;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;
```

Figure 3.2 CUP specification grammar for an example grammar.

```
/* The grammar */
expr_list ::= expr_list expr_part
            | expr_part;
expr_part ::= expr SEMI;
expr      ::= expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
            | expr DIVIDE expr
            | expr MOD expr
            | MINUS expr %prec UMINUS
            | LPAREN expr RPAREN
            | NUMBER;
```

as “Integer” as in the example, but we did not use this feature in our work.

The next section of the CUP specification is for precedences. This optional section helps with parsing grammars that have some ambiguity. Each precedence line gives the identifier precedence, the associativity, and a set of terminals. Precedences listed on a later line are stronger. The possible associativities are left, right, and nonassoc (for nonassociative). The associativities are used to deal with conflicts between equal precedences. The example precedences are all “left”, indicating that if we have, say, a subtraction followed by an addition in the input, then we should do the subtraction (on the left) first. Any terminals not explicitly mentioned in this part of the CUP specification will be given the lowest precedence. The point of these precedences is to help CUP deal with shift-reduce problems. A shift-reduce problem occurs in LR parse tables that are not LR(1). Shift-reduce problems are conflicts between a possible shift action and a reduce action. An example given for the example specification of a shift-reduce problem in the CUP user manual is parsing 3 + 4 * 8. After parsing 3 + 4, CUP would not know whether to reduce the 3 + 4 or shift the * onto the stack. Since * (or multiplication) has a higher precedence than + (or addition), the * will be shifted and the multiplication will be performed before the addition. These declarations also help CUP specify a precedence for each production in the grammar. The precedence of a production is equivalent to the last terminal in that production (with the lowest precedence assigned to a production with no terminals).

The last component of a CUP specification is the grammar itself. In the grammars that we used, we made use of an optional declaration not found in the example grammar that tells CUP which non-terminal to start with. If this start declaration is not provided, then CUP starts with the non-terminal on the left side of the first production listed, which is not necessarily desired. Each production in CUP form has a left hand side non-terminal, followed by "::=", followed by a series of actions, terminals, and non-terminals (with an optional contextual precedence assignment), followed by a semicolon. We did not use actions or contextual precedence assignments ("%prec") in our input grammars. The semicolons are used to terminate a production or set of productions with a given left side non-terminal.

3.1.2 LR Parse Table and Grammar Identifiers

Once CUP has an input specification in the format that it approves, it can be run to generate its output classes. CUP outputs two classes: one which identifies each terminal (and optionally non-terminal) with an integer, and another which contains the parser itself. The parser contains a set of condensed tables detailing the productions, possible actions, and the reductions/ gotos. The parser also contains another class that encapsulates user supplied action code. The user supplied action code tells the parser what to do when the user provides a particular input. CUP provides the command-line option of dumping a representation of the non-condensed LR parse table to the standard output, which we modified to write to a file instead. A copy of the LR parse table generated for our example grammar, as dumped by CUP, is provided in Appendix A. A grammar dump, provided in Appendix B, shows the numerical identifiers of the terminals, non-terminals, and productions. These identifiers are needed to understand the representation of the parse table in Appendix A.

The grammar dump file shows the terminals, non-terminals, and expanded set of productions for our example grammar. Terminals 2 to 11 are the normal terminals that we saw identified in the CUP specification. Terminal 0 is "EOF" or end of file. This terminal is always created by CUP, and serves as a mechanism for initiating or terminating parsing input. Terminal 1 is for identifying errors. The non-terminals are listed from 0 to 5. Non-terminals 1 to 5 are the non-terminals given in the CUP specification. Non-terminal 0 ($START) is generated by CUP to help expand the grammar to have a starting production. The productions 0, and 2 to 11, are those declared in the CUP specification. Production 1 is the extra production CUP created to use as a start production. The input specification for this example grammar did not tell CUP explicitly where to begin (i.e. telling CUP what non-terminal to start with), whereas the specifications we generated for our research did. The start production is given as production 0 in these other instances.

With the productions, terminals, and non-terminals numerically identified, now we can turn our attention to the representation of the LR parse table itself. We always start in state 0. Each state in the action table (the top table) has one or more transitions. These transitions can be shift transitions or reductions. A state can have all shifts, all reductions, or a mix. Each transition identifies the terminal to transition on, the type of transition (i.e. shift or reduce), and then either the state we should transition to next if it is a shift, or the production to reduce on if it is a reduction. Any given state can only have one possible action for any given terminal. We observe that each of the 23 states has some combination of shifts and reductions. We direct the reader to state 21, where the only option is a reduction on terminal 0 with production 1. The way CUP organizes itself, reducing on terminal 0, i.e. EOF, with the starting production, in this case production 1, is the equivalent of accepting a full sequence of input with the LR parser.

After the action table, the reader will observe what CUP has called the “reduce” table, which is just the goto table by another name. The goto table lists transitions as they correspond to non-terminals, and tell the parser what state to go to next.

The goto table is sparser than the action table. Goto transitions only occur after a reduction has occurred. The non-terminals to transition on are determined by the non-terminal on the left side of the production we reduced with. Looking at state 0 in the goto table, we see three transitions: a transition to state 4 on non-terminal 1, a transition to state 1 on non-terminal 2, and a transition to state 6 on non-terminal 3. The action and goto/reduce tables combine to form the action-goto table.

3.2 Definitions

To be consistent with the usual practice in formal language theory, we use the word "string" to refer to a sequence of terminal symbols that can be parsed by the abstract grammar. A string may or may not be terminated by a special symbol that indicates the end of the input. This special symbol cannot appear anywhere else in the string. To be consistent with the tool we use, we refer to this symbol as EOF.

We use the phrase "program input" to refer to a sequence of actual bytes that can be used as an input to a program. Thus, a sequence of bytes would have to be tokenized into a sequence of terminal symbols in order to be parsed by the grammar. Conversely, given a sequence of terminal symbols, each symbol would have to be replaced by a sequence of bytes in order to be translated into a program input.

The following definitions assume that we have a grammar G, the LR parse table constructed for that grammar by a parser generator like CUP, and a parser based on that table.

Definition 3.2.1 A string u is a *valid* string if u+EOF is accepted by G. Otherwise it is an *invalid* string.

An example of a valid string, with literals in place for terminals for readability for our example calculator grammar could be “5 + 6;” whereas an invalid string might be “5 +” since addition requires two quantities to add.

Definition 3.2.2 A string u *covers* state s of the parse table if processing u+EOF causes the parser to pass through state s.

If for our example grammar we have a string of two terminal symbols that takes us from the start state (state 0) to state 3 and then to state 2, then we would say that string covers states 0, 3, and 2.

Definition 3.2.3 A string u is a *reaching string* for a state s if it covers s, but no proper prefix of u covers s.

Using the example immediately above, we would say that our string is a reaching string for state 2, but not for states 0 or 3.

Definition 3.2.4 A set R of strings is a *reaching set* if every string $u \in R$ is a reaching string for some state s. It is *complete* if, for every state s in the parse table, there is a reaching string $u \in R$ for s.

The following definitions assume a reaching set R.

Definition 3.2.5 A state s is a *final* state wrt R if some string $u \in R$ is a reaching string for s and u is valid. Otherwise s is a *nonfinal* state.

Definition 3.2.6 A set S of strings *covers all nonfinal states* wrt R if it contains all strings $u \in R$ that are reaching strings for nonfinal states.

There are three types of states in an LR parse table which do not correspond to non-final states coverage. The first type is the final state. The second type of state is states that cannot be reached by a shift transition from one state to another. The reason for this is that non-final states coverage requires us to follow a sequence of tokens through the LR parse table until we reach a particular state, but strings taking us to non-final states not reachable by at least one shift transition will be the same as some string in the set S that reaches a shift-reachable state. A shift transition will use up a token and then stop, but a reduction followed by a goto will not use up a token. As such, if we take a particular token to activate a reduction and then utilize the corresponding goto transition, we will not "stop" in a state until we eventually shift again, which could be immediately after the goto, or could be some number of reductions and gotos later. Another way of describing this type of state is any state that can only be reached by a goto transition.

The last type of state which we did not include in non-final states coverage was states which could not be reached in any reasonable amount of time. This will be explained more later, but for now, we will simply mention that because the number of possible routes through a table can be infinite, reaching some states is extremely difficult.

Definition 3.2.7 A set S of strings *covers all valid strings* wrt R if, for every string $u \in R$, there is a valid string $v \in S$ such that u is a prefix of v.

Definition 3.2.8 A set S of strings *covers all invalid tokens* wrt R if, for every string $u \in R$, there is a string $v \in S$ such that:

1. v is invalid;
2. $v = uxw$ for some terminal symbol x and string w; and
3. There is a terminal symbol y such that uyw is valid.

In the special case that the reaching set R is complete, these definitions could be expressed in terms of the states directly, rather than relative to R. We are expressing them relative to a reaching set R because, as we discuss later, we were not able to construct a complete reaching set for our two most complex subject grammars.

Strings can cover states not reachable by a shift because the strings used to reach them cause the parser to parse through these states. For example, suppose we have a string that takes us to, and then stops in, some non-final state. From there, some token may cause us to reduce, and then use a goto to enter a state which has some shift transitions. We then shift on this terminal to some other state, using up the input token. However, if we performed the initial reduction with a different terminal, then we will take a different shift transition when we reach the state with the shifts. This will instead take us to a different state than the one reached by the other shift when the input token is used up.

Since parse tables may offer multiple routes to states reachable by shifts, one of these strings, both, or neither may be unique when continued to the final state. In this way, we should expect significantly more valid strings than non-final state strings. Invalid token strings change the first token after going through the state to which the reaching string corresponds, so we should expect these strings to also be more numerous than the non-final state covering strings. Since the sets covering all valid and invalid token strings must be unique to be proper sets, the fact that some of these strings may be redundant means that there will likely be different numbers of strings in each set even though invalid token strings are all based on valid strings.

There is one additional input that we used when testing our programs. We felt that we needed some sort of a benchmark to see how effective basing our input on an LR parse table was, so we created a random input generator too. This random input generator would accept the list of possible terminals, excluding 0 and 1 (EOF and error), and then make inputs of a specified length where the order was fully random (completely ignoring the structure of the LR parse table). The length of the strings was determined by the maximum length of any string (non-final, valid, or invalid) used for that grammar. These strings have the advantage of running error recovery code because they do not conform to the structure of the grammar, but do not get as far into the LR parse table and so may test less of the grammar (and less of the program).

3.3 Algorithms

This section will explain attempts made to use different algorithms to achieve coverage of the LR parse tables. We attempted two different strategies: a basic, depth-first search, and a depth-first iterative deepening search. We discovered that there was a state space explosion problem which prevented us from reaching some of the possible states in any reasonable amount of time. Figures 3.3, 3.4, and 3.5 provide the algorithms we used to obtain reaching strings.

Figure 3.3 Algorithm used to find reaching strings.

```
Algorithm FindReachingStrings(t)
--------------------------------
Input:
- State table t
Output: map of states to reaching strings
Algorithm:
- stateStack = empty stack of states
- str = empty string of symbols
- map = empty map of states to strings
- Visit(t, 0, stack, str, map)
- Return map
```

3.3.1 Description of Algorithms

The algorithm provided in Figure 3.3 maps states to reaching strings. This algorithm calls the Visit algorithm given in Figure 3.4. The Visit algorithm first checks if the stopping condition has been met (to be explained more in a bit). If no string is in the map for the state that Visit is currently at, then the map for that state is set to be the string used to reach the state. Then, after pushing the state onto the stack, for every shift or reduction available in that state, the token used to activate the next transition is added to the current string and the transition is taken. If the transition is a shift, then Visit is called recursively on the state that the shift takes us to. If the transition is a reduction, then the state stack is cloned, a reduction is attempted, and if the reduction was performed successfully, the VisitAfterReduce algorithm given in Figure 3.5 is called.

Once we pop back after trying any given transition, the terminal used to activate the transition is removed from the input string. If all the transitions have been used for a given call of the Visit algorithm, then we pop the visited state off the state stack.

Figure 3.4 Algorithm used to visit states.

```
Algorithm Visit(t, state, stateStack, str, map)
-----------------------------------------------
Input:
- State table t
- State state
- Stack stateStack of states
- String str (string reaching this state)
- Map map of states to reaching strings
Modifies: map
Algorithm:
- If StoppingCondition, then return
- If no string in map for state, then:
   - Set map[state] = str
- Push state on stateStack
- For each symbol x that can be accepted in this state:
   - Append x to str
   - If x induces a shift to nextState:
     - Visit(t, nextState, stateStack, str, map)
   - Else (x induces a reduce on production p):
     - clonedStack = stateStack.clone()
     - success = doReduce(t, clonedStack, x)
     - If success:
       - VisitAfterReduce(t, clonedStack, x, str, map)
   - Delete x from the end of str
- Pop state off stateStack
```

At the end of each call to the Visit algorithm, the state stack is the same as it was at the start of the call. This permitted us to continue the depth-first search after each recursive Visit call had returned.

DoReduce does one reduction step, which may modify the state stack. This is why we needed to clone the state stack before performing a reduction, so that we could restore the state stack by the end of the call to Visit. An attempted reduction might fail because the symbol chosen to reduce with (potentially multiple successive reductions

Figure 3.5 Algorithm used to visit states after a reduction.

```
Algorithm VisitAfterReduce(t, stateStack, x, str, map)
------------------------------------------------------
Input:
- State table t
- Stack stateStack of states
- Symbol x
- String str (string reaching this state)
- Map map of states to reaching strings
Modifies: map
Algorithm:
- If no string in map for state, then:
   - Set map[state] = str
- If x induces a shift to nextState:
   - Visit(t, nextState, stateStack, str, map)
- Else (x induces a reduce on production p):
   - success = doReduce(t, stateStack, x)
   - If success:
     - VisitAfterReduce(t, stateStack, x, str, map)
```

ago) might not be acceptable in the state we are currently in. This would imply that the symbol was not actually valid when we chose it initially.

As was previously mentioned, the VisitAfterReduce algorithm from Figure 3.5 is performed after a reduction is attempted by Visit. This algorithm starts by checking if there is a string in the map to that state, if there is not, the current string is added. Then, if the current input token induces a shift to another state, the Visit algorithm is called on that new state. Otherwise, if the current input token causes a reduction to occur in the current state, doReduce is called again, and if doReduce was successful, VisitAfterReduce is recursively called on the new state that the goto takes us to.

It is possible to execute many reductions in succession with the VisitAfterReduce algorithm. The most significant difference between Visit and VisitAfterReduce is that VisitAfterReduce must work with the input token it is given, whereas Visit tries all of the options available to it.

3.3.2 Stopping Condition

Now we shall discuss our exploration of different stopping conditions, as indicated in Figure 3.4. The first attempted stopping condition was obtained by recording every state visited and then stopping the recursion when we got to a state seen previously. Hence, this initial stopping condition was just that we had seen the state before, as would be expected with a normal depth-first search. Eventually we discovered that this stopping condition was inadequate because it made some states unreachable. These states were unreachable because sometimes it is necessary to pass through a state multiple times in order to reach another state.

A simple example of this is with a production S ::= E + E. This production is for an ambiguous context-free grammar. An ambiguous context-free grammar requires precedence rules to resolve these ambiguities. Most productions are associated with unique states that are reached after the production was successfully used to parse a string. However, because there are two E non-terminals on the right-hand side of the production, we need to pass through the states associated with parsing an E twice before reaching the state associated with the production. In essence, what we realized is that the state space to be explored was not the finite space of parse table states, but rather, the infinite space of stacks of parse table states.

We then tried recording the top N elements of the stack on each call to Visit, and stopping the recursion when we got to a stack in which the top N elements were the same as we had seen before. This is a heuristic; it may prune too much and still prevent us from reaching states, but it is adjustable. A smaller number for N prunes the search space more, and a larger number prunes it less. We therefore referred to this number as the “reaching pruning slack”, or RPS for short. However, we found that when we set RPS to a large number, the algorithm took an infeasibly long time, possibly computing for hours without finishing its depth-first search. We then added a depth bound on the number of recursive calls to Visit, passing this number as an extra parameter to Visit and VisitAfterReduce.

The StoppingCondition thus became either that the depth bound had been exceeded, or that the top RPS elements of the stack had been seen before. Note that when RPS is greater than the depth bound, we are essentially exploring the entire search space up to the depth bound. The end result was a heuristic search algorithm, a depth-first iterative deepening search with multiple stopping conditions. Depth-first iterative deepening works as a depth-first search with an increasing depth bound. The depth bound starts at 0, then increases to 1, then 2, and so on until the goal has been found or an overall depth bound has been reached.

This algorithm was not guaranteed to find reaching strings for all states, but we could adjust the two parameters (RPS and depth bound) in order to maximize the number of states found in a given time. The depth-first iterative deepening also returned more efficient strings (i.e. fewer terminals) than the basic depth-first search since it tried every possible route at each depth. The process used to determine the optimal values for the reaching pruning slack and maximum depth to search to parameters is provided in the next chapter.

The reason that we could not locate every possible state in our LR parse tables was because not every shift, reduction, and goto transition was taken. Simply taking every available shift and reduction at the states reached was insufficient for locating every state. It was required in many cases to go through a state multiple times in order to reach some other states. The need for our stopping conditions arose because we needed to permit this repeating of state visits, yet had to be able to control the amount of time spent searching. This repetition of visits was required to activate not shifts or reductions, but goto transitions.

Before implementing the depth-first iterative deepening strategy with stopping conditions, we dedicated some time to trying to "force" our algorithm to take these goto transitions. If we could direct our algorithm to take these goto transitions, then, at least theoretically if there were no disjoint parts of the table, we should be able to reach any state we wish. The difficulty encountered was that taking these goto transitions requires reducing after taking a specific path from the state with the goto (by shifts, reductions, and possibly gotos from other states along the way). We attempted to direct our route through the parse table when our depth-first search popped back to a state where all of its shifts and reductions had been used up, but some unused goto transitions remained. We considered trying to use the productions themselves to guide this process, but the fact that multiple productions may correspond to one non-terminal (which is what the goto transition would need to be taken on), and the fact that working through a production may involve calling other productions (when reductions along the way are performed) made this infeasible.

We also considered a version of depth-first iterative deepening branching out from the state with the gotos that need to be taken where no pruning of the state space would occur. This would perform an intense, but hopefully short, "local" search until the goto was taken. This proved difficult to implement, and so was not explored further, but could be an area of future research.

Chapter 4

Grammars

This chapter is organized as follows: the subject grammars used for research are introduced, the process used to prepare the grammars and generate their LR parse tables is described, the strategy we used to tune the parameters is explained, and finally, we discuss how we generated test inputs.

4.1 Subject Grammars

The grammars used for research were BC, HTML, XML, and URL.

The first grammar, BC, stands for basic calculator, and is a grammar for a Unix utility used to perform calculations. We obtained the BC grammar from [27]. In this form, we were provided with the possible tokens used in the grammar as well as their

Table 4.1 Basic data for each grammar, the number of strings generated for each coverage type, and the maximum depth of tokens and reaching pruning slacks used to garner those strings.

Grammar	**BC**	**HTML**	**URL**	**XML**	**Example**
Terminals (including EOF and error)	35	441	41	177	12
Non-terminals (including START)	17	164	59	132	6
Productions (including START)	65	442	180	229	12
States in LR Parse Table	126	807	249	390	23
States Reachable by Shift	78	440	108	152	11
Shift Transitions	512	2982	713	456	46
Reductions	449	16499	1400	1041	63
Goto Transitions	95	2464	283	378	12
Non-Final Strings Generated	75	354	53	129	9
Invalid Token Strings Generated	108	449	131	8	19
Valid Strings Generated	101	376	158	6	14
Total Strings Generated	284	1179	342	143	42
Maximum Depth of Tokens Used	46	35	36	89	24
Reaching Pruning Slack	2	3	9	4	3

corresponding literals, the structure of the grammar in a format similar to but not quite the same as what CUP expects, and the rules of precedence and associativity for the operators. The syntax of BC is similar to that of C. BC is a considerably larger grammar than the example one from the previous chapter, but is still significantly smaller than the other three used for research. As a whole, BC was also one of the simpler grammars to work with.

The next grammar, HTML, stands for HyperText Markup Language, and is the standard mark-up language used in creating web pages. We had a copy of a grammar for HTML from previous work that we were able to use with some adjusting. This grammar was a BNF grammar created by Marek Gryber (obtained from http://www.angelfire.com/ar/CompiladoresUCSE/COMPILERS.html) based on the official HTML grammar (from http://www.w3.org/TR/1999/REC-html401-19991224/) [5]. HTML was more complex than BC, but less so than URL or XML. The main difficulties unique to HTML arose due to the sheer size of the grammar. Table 4.1 shows that the number of productions used in HTML is 442. The next largest grammar is XML with 229. For comparison, the example grammar discussed in the previous chapter has only 12 productions. Additionally, the HTML and XML grammars in their basic form had literals rather than terminals. These literal strings needed to be exchanged for terminals to have either grammar accepted by CUP.

The grammar we used for URL was technically a grammar for URIs, as defined in Appendix A of [4]. A URI is a uniform resource identifier, whereas a URL is a uniform resource locator [4]. All URLs are URIs, but not all URIs are URLs, as there is also a category of URIs known as URNs (uniform resource names) separate from URLs [4]. URI is a more general term than URL, but our plan for this thesis was to test URLs as best we could, so we continued to refer to the grammar as URL, even if technically everything is done with the URI grammar. We did not find a proper grammar for URLs alone.

XML stands for Extensible Markup Language, and is a grammar used to define a set of rules to encode documents in a way readable to both computers and people. As previously mentioned, XML was the second largest grammar in terms of productions, but more significantly, was the most difficult to generate strings for that could reach a final state than any of the other grammars. Like with HTML, we had a copy of the grammar available from previous work. This XML grammar was a BNF grammar published by W3C (obtained from http://www.w3.org/TR/xml11/) [5].

4.2 Subject Grammar Preparation

There were three basic steps that needed to be performed for the subject grammars before we could generate any strings from them. The first step was to put the grammar into a format acceptable to CUP. The second step, which could be done at any time, was to create a Java class called "scanner." CUP generated classes for the third step that could not compile without a scanner. The third step was to use CUP to generate the LR parser and symbol identifying Java classes.

As was previously mentioned, we needed to create a scanner for each grammar to make the parsers compile. Since we required four scanners, and each one was fairly simple in set up yet time consuming to make, we automated the scanner-making process. However, what we placed in these scanners changed greatly as we went through the process of programming an algorithm to generate strings. CUP's initial model scanner was intended for different purposes than what we intended, as it was meant for scanning actual input to run through the LR parse table and we wanted to use the LR parse table to lead input generation. The scanner morphed into a class that would return the next terminal to our algorithm as the terminals were requested by searching the table. By the end, the scanners were mere shells included for the sake of making CUP compile, performing no actual creation or handling of data.

4.2.1 Grammar Preparation

Now we shall discuss some specific issues encountered while preparing the subject grammars for use by CUP.

To make BC suitable for use with CUP, we needed to make some minor adjustments to the format of the grammar. One such adjustment was to replace some of the literals with representative terminal names. There were few such literals, mainly parentheses and semicolons, so this was done manually. We also were able to accept the precedences provided by BC and adjust them to be suitable for CUP without incident. We wrote a simple Java program to convert the adjusted BC grammar into a proper CUP specification, and used it on all four of the research grammars once they were in a suitable state. This permitted us to use CUP to generate the Java classes needed to work with the LR parse table. Overall, BC was by far the easiest of the four grammars to prepare for CUP.

CUP does not have a means for handling literals directly, so we needed to convert all of the hundreds of literals into terminals for HTML and XML. We decided to automate a process to use on both grammars that would permit translating between terminals and literals. Whereas BC or URL required us to choose randomly from a list of possible literals that could correspond to some terminals, since HTML and XML provided us with literals but no terminals, we decided to assign a different terminal to every literal for simplicity. If we switched out the literals for terminals in the grammar, we could create strings. If we replaced the terminals with literals from these strings, then we would have appropriate input for our subject programs.

We employed a three step process in handling our literal-terminal correspondence issue. The first step involved making a first pass over the grammar (either HTML or XML) and producing a file identifying each literal with a terminal name. The terminals were given names in the form LITERAL<number of literal as appears in grammar>. Here is an example of one line in the file created by the first pass over HTML:

```
LITERAL39:<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http:
//www.w3.org/TR/html4/strict.dtd">
```

The second step in the process was to perform a second pass over the grammar file, this time using the grammar and the file from step one to create a grammar in the format that CUP expects so as to create a CUP specification. The last step was to turn completed strings for HTML or XML into acceptable program inputs, but this will be described later.

One minor issue discovered when trying to create the CUP specification for HTML was that the grammar has a non-terminal named "code." CUP has a list of reserved words that cannot be used for terminals or non-terminals and code is one of them. Before we could get CUP to accept our specification for HTML, we needed to rename the non-terminal code in all instances. Non-terminals do not appear in the strings, so this renaming had no lasting effects.

The main problem with preparing the URI grammar early on was that it was not presented as a context-free grammar. Some minor difficulties similar to what was experienced with BC in terms of literals were easy enough to overcome, but the grammar itself had to be rewritten to remove ambiguity. This proved challenging, and could have potentially introduced problems later if any mistakes were made. Whether due to any mistakes in the conversion or just the way the URI grammar was set up in the first place, errors continued to prevent CUP from accepting the input (once it had been automatically adjusted to be in CUP specification form). The CUP program in charge of evaluating the specification reported many shift-reduce conflicts. A series of precedences were placed in the CUP specification in an attempt to combat this problem. Unfortunately, unlike BC, the URI grammar did not provide us with the correct precedences, so we were left taking our best guess. The presence of the precedences reduced, but not did completely remove, the conflicts. We then used a command-line argument offered by CUP which would tell the CUP program to expect and accept a particular number of conflicts. We used this argument and CUP compiled, though it still left us with 77 warnings. HTML and BC had no warnings, but the example grammar had two (and no precedence issues) and XML had 36. This indicates some of the warnings for the URI grammar may be unrelated to the conflicts.

4.2.2 Parse Table Generation

Once we had a CUP specification ready for each grammar, we ran CUP to generate the LR parse tables and lists of terminal identifiers. As shown in Table 4.1, the number of states in each parse table was roughly proportional to the size of the grammar it was created from. The HTML grammar's LR parse table has over 800 states. XML has the next highest with less than half of that number, with BC and the URI grammar having 126 and 249 respectively. For comparison, the example grammar found in the appendices has only 23. We also note the number of states reachable by a shift for each grammar in Table 4.1. This number provides a ceiling for the number of non-final state reaching strings generated for each grammar, once final states are subtracted.

XML had fewer shift transitions and reductions relative to its size compared to the other grammars. This could be an indicator that the LR parse table corresponding to the grammar is more complex, since more transitions means more options for getting from one state to another.

There was an issue with the LR parse table file for HTML. HTML's massive size meant that the action table initialization in the LR parse table class generated by CUP did not compile because it was "exceeding the 65535 bytes limit." To make it compile so that we could use the table, we needed to initialize the action table in several smaller chunks, and then combine those chunks to form the full action table. The action table for HTML was the only instance of this issue.

4.3 Parameter Tuning

As was mentioned earlier, obtaining reaching and valid strings for all states for the larger grammars used in research was difficult due to the possibility of infinite paths through the LR parse table that would need to be traversed. We could find more states with reaching strings by running the depth-first iterative deepening to a greater maximum depth, but it would also be significantly slower. Additionally, by increasing the value of the reaching pruning slack (RPS), the number of states reached might increase depending on the structure of the grammar, but again, the time would increase too.

We decided to run our depth-first iterative deepening algorithm for finding reaching strings on all the grammars with many combinations of maximum depth and reaching pruning slack to determine the optimal settings for these parameters for each grammar. We tried all values for RPS from 1 to 10 and values for maximum depth up to 100. We cut off investigating a particular combination of RPS and maximum depth if the run of the algorithm with depth 1 less required greater than 5 minutes to run.

We obtained the data needed for determining the optimal values of reaching pruning slack and maximum depth to search to through the use of shell scripts. The scripts performed the procedure in Figure 4.1.

Figure 4.1 Script procedure for obtaining RPS and maximum depth.

```
- For each of the grammars:
  - For reaching pruning slack from 1 to 10:
    - Set maximum depth = 5
    - While the maximum depth <= 100, or the previous run at this
      reaching pruning slack value took more than 5 minutes to complete:
      - Run the algorithm to achieve reaching strings on the grammar
        for the given values of reaching pruning slack and maximum depth
      - Increment the maximum depth to search to by 5
```

We stored this data in CSV files, one per grammar. We then wrote a script in the statistical programming language R to turn this data into graphs that could be used to visualize the optimal parameters more clearly. The R script plotted two graphs for each grammar's CSV file. One graph plotted the points in the CSV file by comparing the different values for the maximum depth searched to the time spent searching. A different line connecting the points was plotted for each value of the reaching pruning slack. The other graph compared the maximum depth for each value of reaching pruning slack to the number of states reached. From these graphs, we were able to make educated decisions about the optimal values for the reaching pruning slack and maximum depth parameters. Table 4.1 shows the maximum depth and RPS we decided on for each grammar. The maximum depth is slightly higher in this table than the numbers shown in the graphs because they include the additional depth required to search for a final state after reaching the non-final states (to generate valid and invalid strings).

Figure 4.2 shows for BC the time spent searching for states compared to maximum depth, and Figure 4.3 shows the number of states found compared to maximum depth. In both graphs, each line corresponds to one value of RPS. Figure 4.2 shows an exponential spike in the time spent searching for each value of reaching pruning slack. The time spent searching for RPS of 9 and 10 do not even show up after a depth of five because the operations timed out (which is not the same as merely going over 5 minutes). It may be that due to its simpler nature, BC tended to generally have smaller stacks when searching for states. If the stack is sufficiently small, say, less than or equal to 9 or 10, providing an RPS of a similar value would have little to no effect on pruning the search space. This would cause the algorithm to search through the LR parse table, trying every possible path. Figure 4.3 demonstrates that a reaching pruning slack of 2 or higher is sufficient to find all of the states possible for a reasonable value of maximum depth. There is no reason to apply greater values for the same result. The maximum depth for BC is 46 and the maximum number of states could be found with an RPS of just two. This value of RPS is the lowest for all of the grammars, even including the example one. The maximum depth of 46 is not unreasonable, but is greater than those for all but XML.

Figures 4.4 through 4.9 show similar graphs for HTML, URI, and XML. HTML seems to have a jump in the time spent searching when going from an RPS of 3 to 4, as shown in Figure 4.4. However, as Figure 4.5 shows, this value (of 3) also reaches the greatest number of states. Hence, it makes the most sense to use an RPS value of 3 for HTML. HTML was successful with a very reasonable maximum depth of only 35 (especially considering the size of the table), so we accepted that value for the depth parameter.

For the URI grammar, the increase in time for each incrementally greater value of RPS or maximum depth appears somewhat consistent in Figure 4.6. We can see that each increase in RPS increases the number of states found up until the last few values in Figure 4.7. This would indicate that a large value for RPS is necessary, though a large number of states can be found with a limited maximum search depth. As such, the values we determined would find the most states in the most reasonable amount of time were a RPS of 9 and a maximum depth of 36. 36 is a reasonable value for depth, only slightly higher than what was used for HTML. The reaching pruning slack of 9 was far greater than that of any other grammar though. The reason for this high value may have been because some of the productions for the URI grammar had many symbols on the right-hand side, resulting in needing an extended period of searching before one could reduce properly on those productions and take the corresponding goto. This means that the reachable states could be found rather quickly with the URI grammar, but a lot of freedom was necessary for finding these states.

XML has the most gradual increase in required search time for increasing values of depth and RPS, as Figure 4.8 shows. However, Figure 4.9 indicates that for moderate values of RPS, the number of states found continually increases until the value of maximum depth searched is quite large. XML was best run with an RPS of 4 and a maximum search depth of 89. 89 is a far greater depth than what was required for any of the other grammars. This would indicate that XML continued to have states found in the parse table as we search to increasingly greater depths, rather than plateauing quickly the way the other grammars did. Curiously, due perhaps to the complexity of the grammar, an RPS of 1 finds almost no states.

4.4 Test Input Generation

In the previous chapter, we discussed how a depth-first iterative deepening search algorithm was used to find sets of strings providing non-final state coverage, invalid token coverage, and valid coverage of an LR parse table. We illustrated these concepts with a simple example grammar. As the statistics show in Table 4.1, the grammars we performed our research with were significantly more complex than the example grammar. These grammars have more terminals, non-terminals, states, transitions, and productions than the example grammar.

Note, as mentioned in the previous chapter, that the number of strings generated in each category for each grammar is dependent both on the difficulty in getting through the LR parse table, and the elimination of redundant strings. The number of non-final strings must always be equal to or less than the number of states in the table reachable by a shift transition. Any other states do not use up a token when passing through the state, so their strings will be duplicates of the string that reached the last state reached by a shift. However, for invalid token and valid strings, more strings are possible because the parser can take other routes through the non-shift states and thus count these routes as new strings to reach the final state. There tend to be more invalid token strings than valid strings because substituting a valid token by a random invalid one reduces the chances of redundancy since there tend to be more invalid tokens than valid ones for any given state.

With the optimal values for RPS and maximum depth achieved, we applied these parameter values to create the non-final, invalid token, and valid strings. We also used the lists of terminals to generate the strings of randomly ordered tokens to use as our benchmark. Once we had generated all of these strings, we converted them into input that could be parsed by the programs being tested. We discuss some of the particulars for the different grammars below.

For BC, we were able to generate 75 strings leading us to non-final states, which was good considering there were only 78 shift states in the first place. We also generated 108 invalid token strings and 101 valid strings, as shown in Table 4.1. This gave us a total of 284 strings generated for BC, where the maximum depth of terminals in the strings was 46. We used these numbers to provide a fair setting for our benchmark string generating approach. We created 284 strings for our benchmark approach, giving it a total equal to all of the other types of strings combined to ensure a fair playing field in terms of obtaining code coverage. Also, every string was made up of 46 terminals, whereas only our longest strings were that length for the LR parse table strings. This means that the benchmark approach would again have the advantage by having more input per string with which to stimulate code coverage. The results of the competing approaches are shown in the next chapter.

As previously indicated, the last step for obtaining program inputs was translating the 568 (LR parse table and benchmark) BC strings into sequences of bytes that could be fed to the subject program. This involved not only translating the handful of literals that had been made into terminals initially when preparing the grammar back into literals, but also choosing literals for some terminals which correspond to multiple possible literals. The BC grammar in its natural state has literals named MUL_OP, ASSIGN_OP, REL_OP, and INCR_DECR that stand in for multiple possible tokens. For example, INCR_DECR can have substituted for it either —or ++. We wanted to give all of the options for these literals a fair shake, so we set up a program that would substitute any appearance of the terminals with a randomly chosen literal from those corresponding to it. Beyond these specific terminals with pre-defined lists from which to choose literals, BC also has other terminals that would need to be substituted by suitable randomly chosen literals from no pre-defined lists. An example of this would be substituting the terminal NUMBER by an actual numerical value. We chose several reasonable literals for each of these terminals, and again, randomly chose a single instance each time a substitution needed to be made when translating from a string to a program input.

We were able to generate 354 non-final state strings, 449 invalid token strings, 376 valid strings, and 1179 random terminal choice strings for HTML. After generating this huge number of strings, we needed to take the file identifying each literal with a terminal name and translate all the terminals back into the corresponding literals to produce proper HTML program input. This was the final step in the three-step process used to translate back and forth between literals and terminals for XML and HTML mentioned earlier.

We generated 158 valid, 53 non-final, 131 invalid token, and 342 random terminal choice strings for the URI grammar. The 53 non-final strings are less than half the maximum number possible on the basis of states reachable by at least one shift transition. The other grammars were all over 80%. This shows the difficulty of reaching any particular non-final state for the URI grammar, though the valid and invalid token strings seemed more reasonable in number. The strings were converted to program input in a manner similar to that of BC, since some literals could correspond to a choice of terminals.

We were able to generate strings to reach a respectable 129 non-final states for XML. Generating valid and invalid token strings for XML however was a disaster. As Table 4.1 shows, we could only generate 8 invalid token strings and 6 valid strings for XML.

To be approximately proportional to the other grammars, we would probably expect about 20 times that many. This showed that our algorithm would tend to get lost in the LR parse table, as it could find a reasonable number of non-final states, but then could not locate the exit by completing the strings so as to reach the final state. While it is possible that some outputs could simply have been removed due to redundancy if they all needed to trace a particular path to the final state, having this few for so large a table must still be an indicator of the challenging nature of this grammar. XML well demonstrates the state space explosion problem of having many potential paths through an LR parse table but only a small number of those paths being useful.

Once we generated the non-final, invalid token, and valid strings for XML, along with 143 random terminal strings with a length of 89 literals each, we were able to run the third step of our literal-terminal switcher to create program inputs. Some of these inputs were short, but most were extremely long due to the difficult nature of XML. That some parts of the grammar could be potentially disjoint, or that there was a low average number of transitions per state, could have led to the difficulty in generating strings that reached the final state.

Figure 4.2 Time required for search versus depth searched for BC.

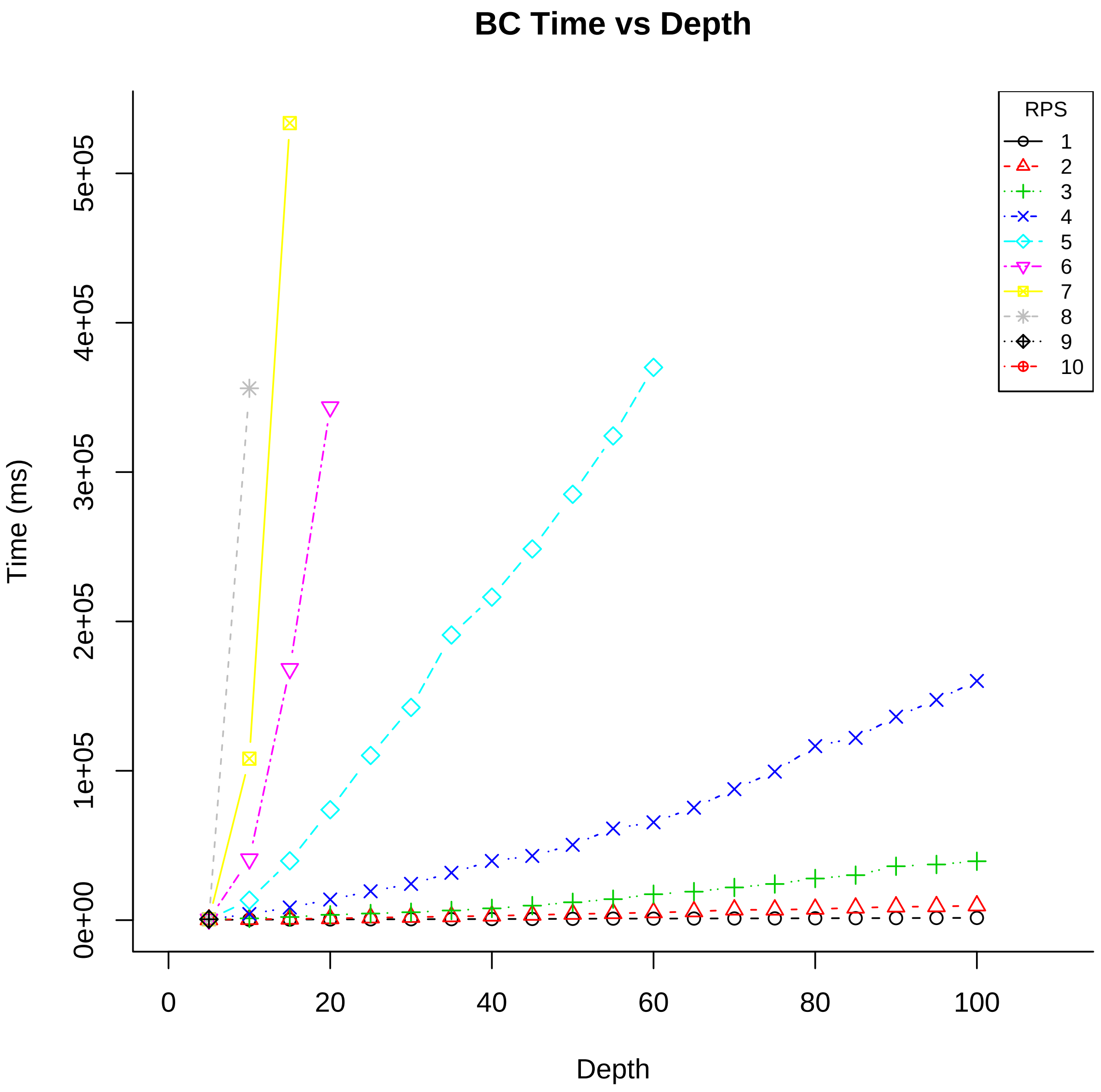


Figure 4.3 Number of states found versus depth searched for BC.

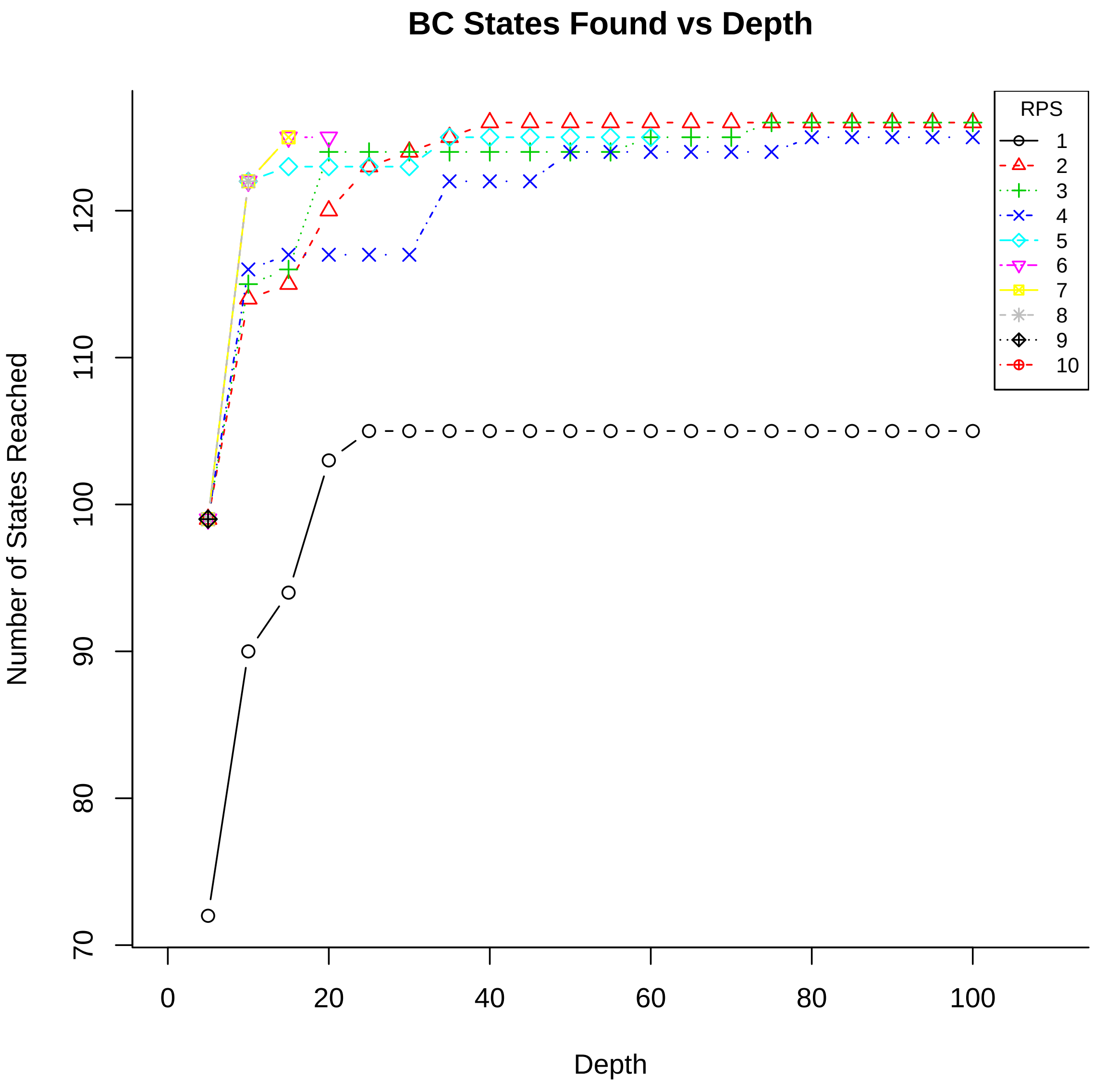


Figure 4.4 Time required for search versus depth searched for HTML.

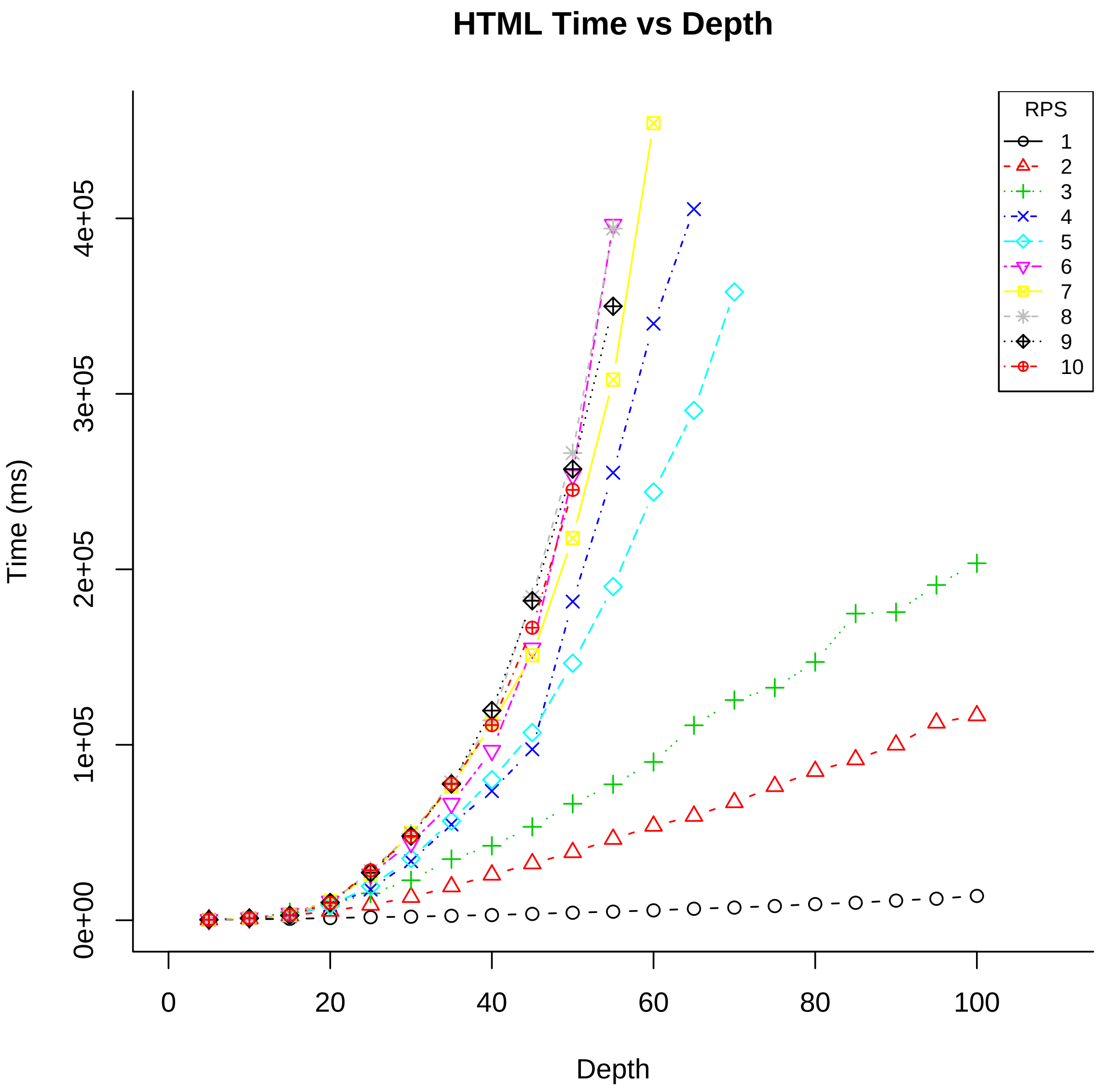


Figure 4.5 Number of states found versus depth searched for HTML.

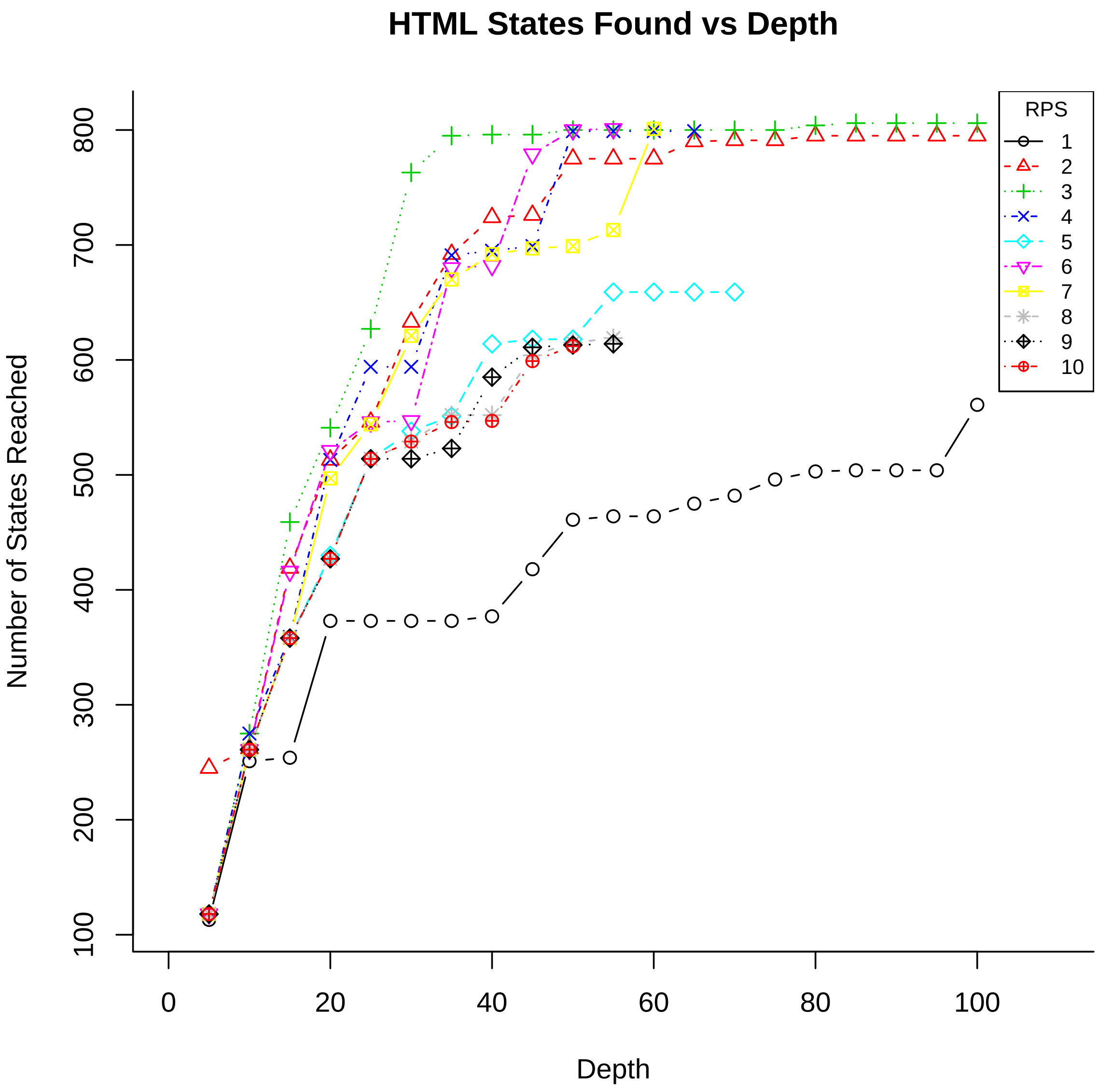


Figure 4.6 Time required for search versus depth searched for URL.

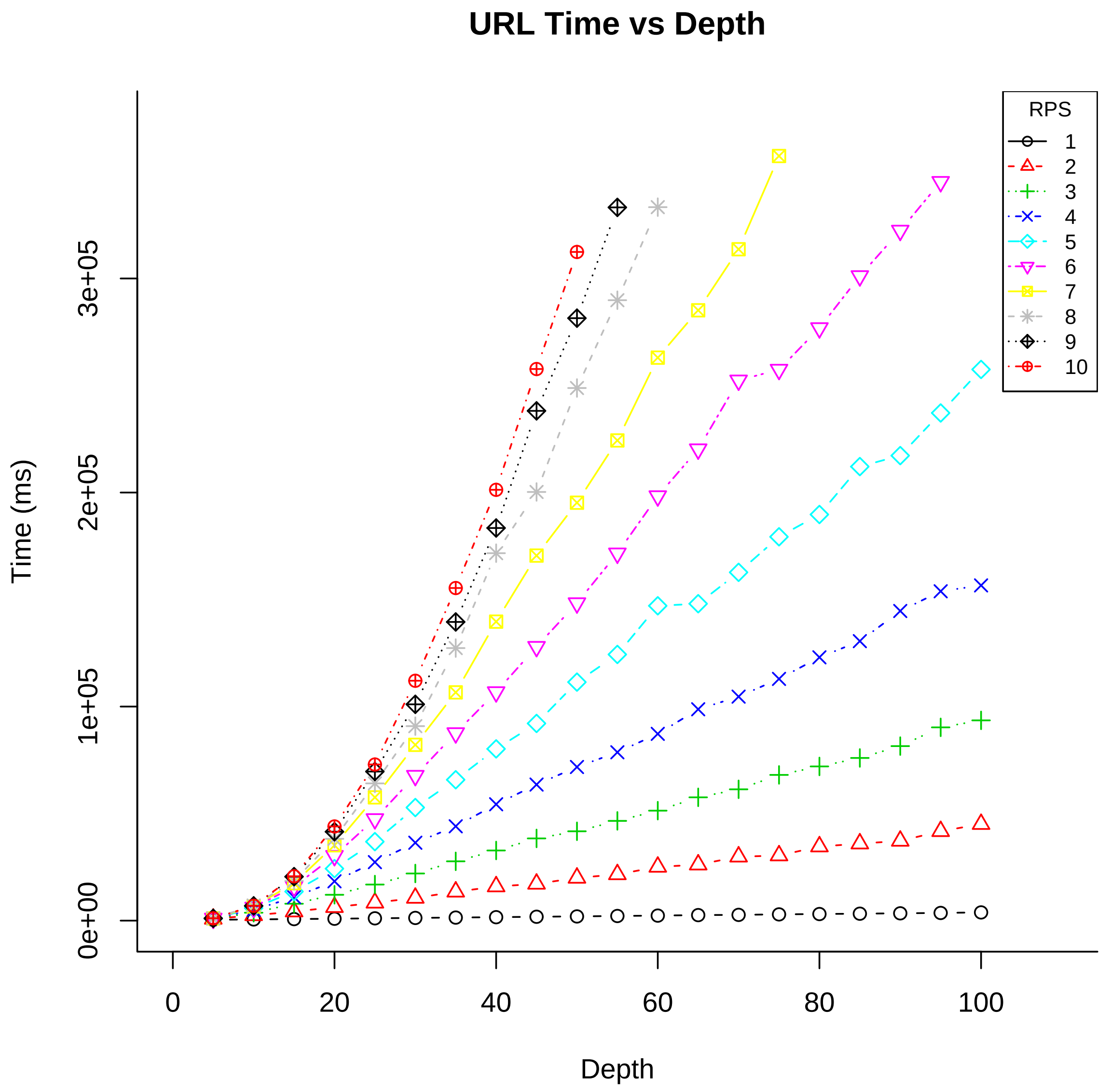


Figure 4.7 Number of states found versus depth searched for URL.

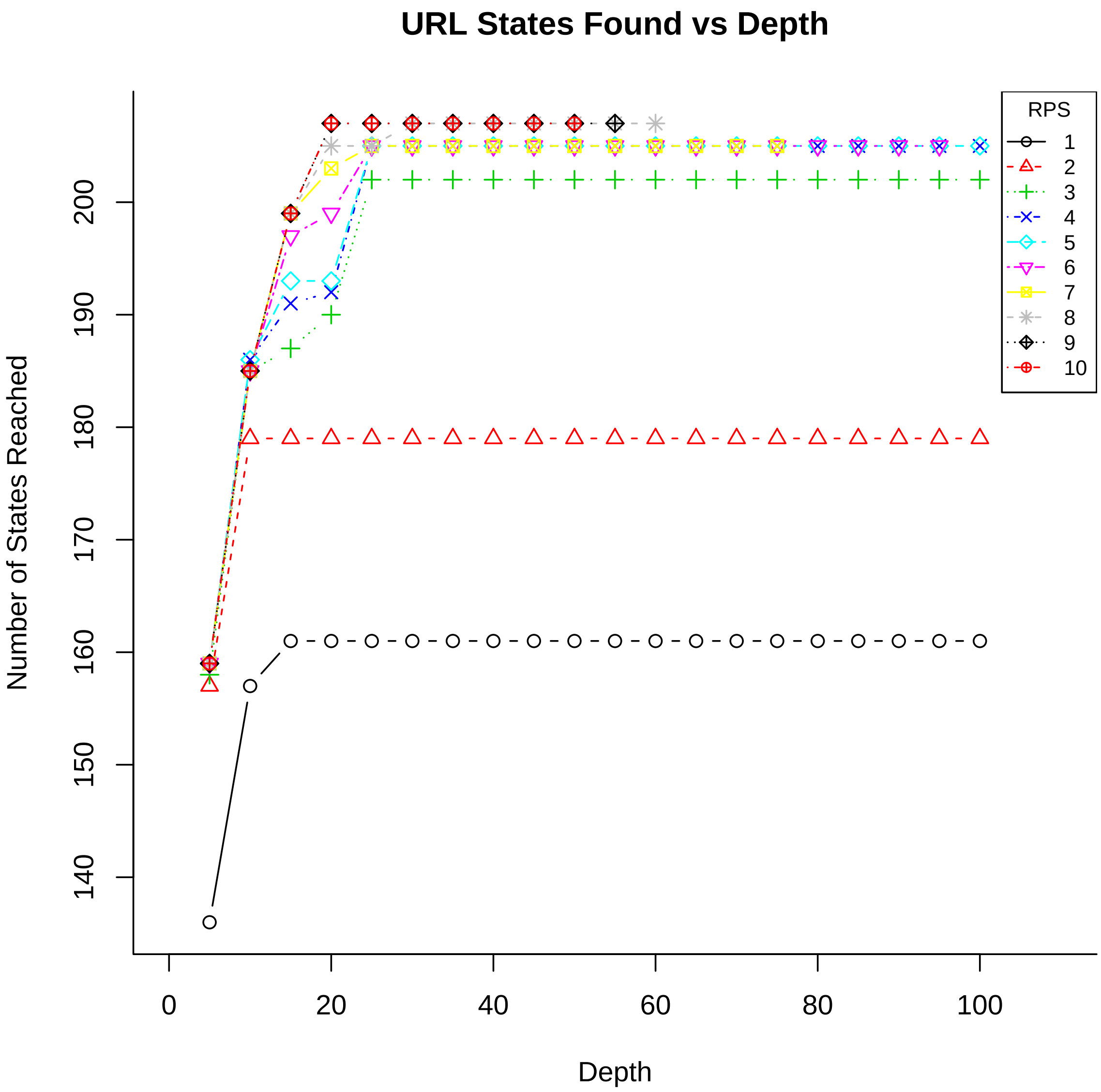


Figure 4.8 Time required for search versus depth searched for XML.

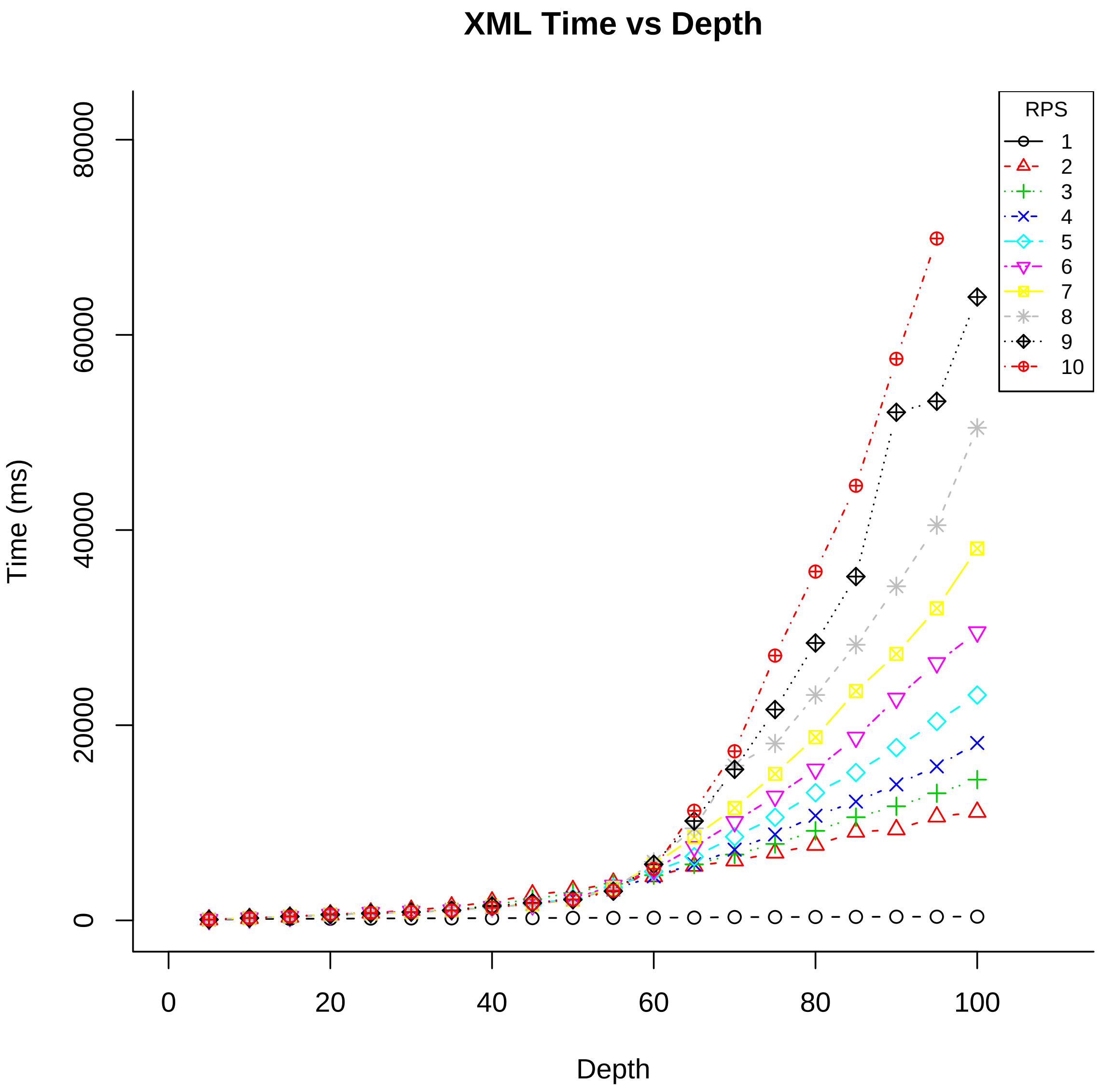
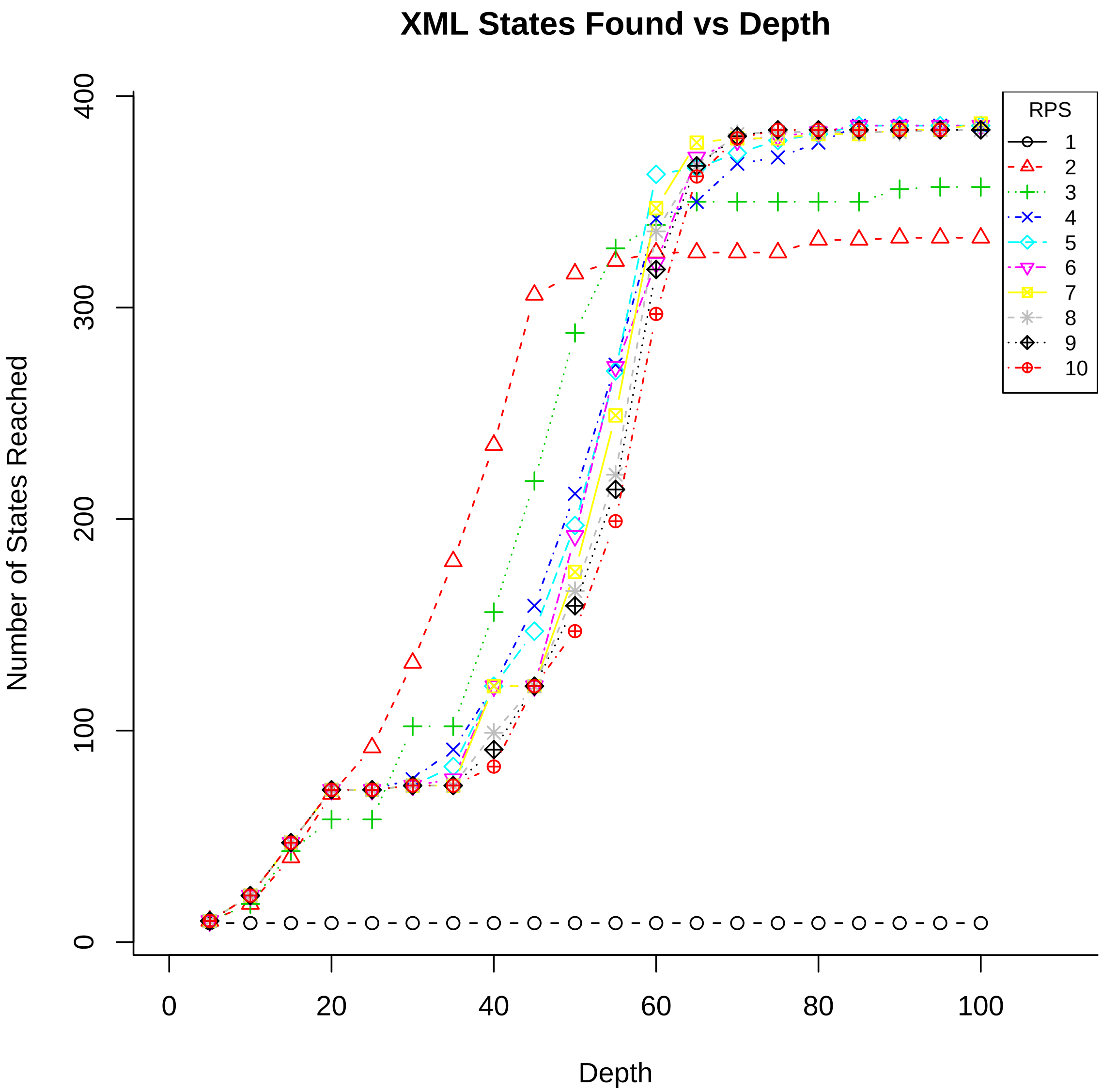


Figure 4.9 Number of states found versus depth searched for XML.



Chapter 5

Application and Results

The outline of this chapter is as follows: we provide a description of the subject programs, mention how the programs were prepared for use, detail the procedure used to run the inputs on the programs, discuss the code coverage results, and analyze the test results.

5.1 Subject Programs

The subject programs that were used for testing on were `bc`, Jtidy, URI.java, and NanoXML. Each of these programs corresponds to one of the subject grammars.

The `bc` program is the calculator that the grammar BC is based on, so it is the most reasonable option to test with [27]. `bc` stands for “basic calculator,” though it is often referred to as “bench calculator.” `bc` is “an arbitrary precision calculator language,” and is generally used as either an interactive mathematical shell or as a mathematical scripting language. The program `bc` is written in C, and has a syntax similar to C. We looked for a reasonable substitute written in Java to use the same tools for code coverage but could not locate anything acceptable. We used `bc` version 1.01.

Jtidy is an HTML syntax checker and pretty printer written in Java, and was used for testing strings created from the HTML grammar. Whereas the `bc` program required inputs that exactly followed the BC grammar, Jtidy has a different function with respect to HTML. Jtidy accepts HTML input, and then, if possible, corrects and cleans it up. This is still a completely valid instance of a need for grammar-based testing, and so this permits us to see how our approach performs in situations in the field other than programs that simply accept and follow the instructions provided in the input. This could mean that erroneous input will stimulate more code for Jtidy compared to a program like `bc`, since Jtidy actively attempts to fix the HTML input provided. Valid input may be of less interest to Jtidy, since Jtidy would have no action to take with it aside from checking its validity.

NanoXML is a simple and fast XML parser written in Java, and was used for parsing inputs from the XML grammar [12]. NanoXML was first released in April of 2000 as a spin-off of another project. NanoXML 2 was released in 2001, which contained more features than the first version, but remained small and efficient as parsers go. NanoXML was intended as a simpler parser than SAX or DOM. We used the current version of NanoXML, version 2.2.1, which was released in April of 2002.

URI.java is the Java class used to create Uniform Resource Identifiers. URI.java represents a URI reference as defined by RFC 2396 [4], which was where we developed our context-free grammar from for URLs (or more accurately, URIs). We needed to do some investigating to determine what program would best test the URI inputs that we created. The subject programs for the other grammars had all been clear from the start, at least once finding an adequate Java replacement for BC was ruled out. There are multiple programs that accept input in the form of URLs or URIs available, but we needed one that would test the structure of the inputs to perform meaningful grammar-based testing. We decided that the best option was to create instances of the URI Java class because it claimed to analyze the validity of the URIs as they were created according to the same RFC 2396. There were some deviations of this class from the RFC, but we felt it was close enough to work for our purposes. Even though our starting plan for this thesis had been to work with URLs, it made more sense to test with the URI class since that was the basis for our grammar. Our subject program was only one class, URI.java, which meant that when we evaluated code coverage, the differences in lines of code covered would be less dramatic.

5.2 Subject Program Preparation

In this section we discuss the steps taken to prepare the subject programs to receive our generated inputs. After unzipping and organizing the subject programs, the first step was to compile all of the relevant code for each program. We needed to compile 10 files for `bc`, 53 for Jtidy, 24 for NanoXML, and 2 for URI (the second one for URI will be explained in a bit). After compiling the code, we instrumented the classes we wished to measure code coverage on. We instrumented with Cobertura for Jtidy, NanoXML, and URI.java because they were written in Java, and instrumented with `gcov` for `bc` because it was written in C. To prevent needing to constantly recompile and instrument the programs, we zipped all of the compiled code and the file declaring which files had been instrumented for each program into zip files (one zip file per program). This way we could just unzip the compiled and instrumented files fresh each time we needed them, and then delete them after running the program with a set of inputs.

There were a number of difficulties encountered in preparing each subject program to accept inputs. We shall discuss a few of them here.

After compiling the `bc` subject program, we determined that the program would not terminate automatically each time an input was run to completion. To overcome this difficulty, we made one additional input file whose contents consisted simply of the word “quit” so as to force `bc` to close down and start up again with each input.

When running Cobertura on Jtidy, we discovered that the coverage was appearing as “Not Applicable” for all classes. This was because Cobertura requires that code be compiled with “debug=true.” While this setting is the default for compiling by the command-line, ant files used to build projects, like the one Jtidy provides to compile all of its code, implicitly have the default set to “debug=false.” We needed to explicitly change all of the lines performing compilation of Java source files in the Jtidy ant file to state “debug=true.”

There were three main non-standard difficulties encountered with compiling NanoXML. The first was that since NanoXML is an older program (written in 2002), there were conflicts between the version of Java available to us and the version that NanoXML expects. Once this was accounted for, we discovered that this change of versions rendered the reserved word “enum” unusable, which required renaming all instances of variables named enum throughout NanoXML. Then, we discovered that since compiling NanoXML uses “unchecked or unsafe operations,” we needed to add some command-line arguments to suppress checking the safety of these operations when compiling.

URI.java does not have a main method to be run and have inputs fed into it. Hence, in order to test this subject program, we needed to first write another Java class that would request URIs and then try to test if the syntax was valid with URI.java by creating instances of URIs. Our program would read a file whose name was provided on the command-line, then attempt to create a URI from the string read in as an argument to the URI.java constructor. Our program printed out if a given URI was valid, but allowed the main method to throw whatever exception it encountered so that we could get an accurate program output result. This created Java class was not included in code coverage calculations.

5.3 Procedure

This section will explain the procedure used to apply inputs to the subject programs.

Upon obtaining all of the non-final, valid, invalid token, and randomly generated benchmark strings and then converting those strings into bytes that could be fed to programs, the next step was to set up those program inputs such that the programs could be run on them. We created directories for all of the different collections of inputs, and then placed each input into its own text file within these directories. These files were named by the order in which they were created. We used shell scripts to run the subject programs with these collections of input text files. We had two shell scripts to run the inputs for each subject program. We had a high level script that would set up the subject program and tell the program which directories of inputs to run, and a lower level one that would handle running the inputs. The general procedure for these scripts is provided in Figure 5.1.

We wanted to evaluate the contributions to code coverage of each kind of input, and the degree to which they overlapped, so we ran all the test inputs, but also subsets of the inputs that left out one or more groups of inputs. Since there are four directories of subject inputs for each possible subject program (namely, non-final, valid, invalid token, and benchmark), there were 15 possible combinations that needed to be run. These combinations are each of the four directories run individually, each combination of two directories, each combination of 3 directories, and all four directories run together. The full set of line coverage and branch coverage outputs for each of these 15 combinations are provided later in this chapter in Figure 5.1.

The standard output and standard error for each directory was used to evaluate how many inputs the subject programs correctly received as being either valid or invalid. We did not make use of the outputs for all 15 combinations. We only needed the outputs and errors for the non-final, valid, and invalid token runs.

The code coverage output location that Figure 5.1 mentions varies, depending on whether we are evaluating code coverage with `gcov` or Cobertura. Cobertura outputs many files, and so the scripts for Jtidy, NanoXML, and URI.java provide a different directory for each output location. `gcov` outputs all of its data as a single set of statistics, and so for `bc` we simply redirect the coverage data for each combination of inputs into its own text file.

The procedure described in Figure 5.1 was run in the Unix environment. Since hun-

Figure 5.1 General procedure for running inputs on subject programs.

```
-Begin in top level directory for subject program
-Move to location in program directory required for execution
-Remove leftover files or directories from previous runs
-For every combination of subject inputs for the subject program:

    -Select code coverage output location
    -Unzip file containing compiled program files and coverage map
    -For all directories of subject inputs to run in the current subject
     input directory combination:

        -For each file in the current input directory

            -Run the subject program on the current file in a way so as
             to induce code coverage and send both the standard output
             and standard error to an output file

    -Enter all coverage data into code coverage output location
    -Remove unzipped code and coverage mapping files from directory

-Return to top level directory for subject program
```

dreds, or even thousands of inputs might need to be run for the many combinations to the subject programs, the time required to execute these scripts could be measured in hours or days. We discovered that the Unix terminal would tend to time out and disconnect when run for this long, causing the terminal to send a “HUP” (hang-up) signal that would disrupt execution of the scripts. To combat this issue, we used the Unix `nohup` command so that the processes would continue running even if the terminal times out. `nohup` is a POSIX command that tells the terminal to ignore the HUP signal. The `nohup` command was especially critical for running the scripts on Jtidy since HTML generated so many inputs.

5.4 Code Coverage

As was previously mentioned, we used Cobertura and `gcov` for obtaining code coverage on our subject programs. Cobertura is a free Java tool that not only computes the values of code coverage (line and branch), but also shows the user precisely which lines of code have been covered. Cobertura is based on the tool jcoverage and was initially programmed by an employee of the company that wrote SAS, a statistical software. `gcov` is a tool to be used with `gcc` to test code coverage. `gcov` computes its values for branch coverage in a different way than Cobertura, which is why we placed most of our focus on line coverage in this thesis so as to keep our data comparable. `gcov` provides the user with exact counts of the number of times each statement in a program is executed. `gcov` computes the percentage of the total lines of code covered in each file, so we needed to calculate the number of lines covered (as an integer rather than a percentage) manually. We obtained the amount of line and branch coverage for each class, the aggregate values for each package, and the total values for each full program for each set of tests.

We looked for a correlation between the number of inputs of each category (minus benchmark) and the amount of coverage. We took the number of inputs in each category for each grammar, normalized them to their grammar, and measured the correlation against the normalized number of lines covered. We observed a correlation of 90.4%, which is quite strong. However, if we remove the three outlier values (valid and invalid token inputs for NanoXML and non-final inputs for URI.java), the correlation drops to 34.1%. This means that if there is a dramatic difference in the number of inputs then an effect will be observed, but otherwise it is not a reliable predictor.

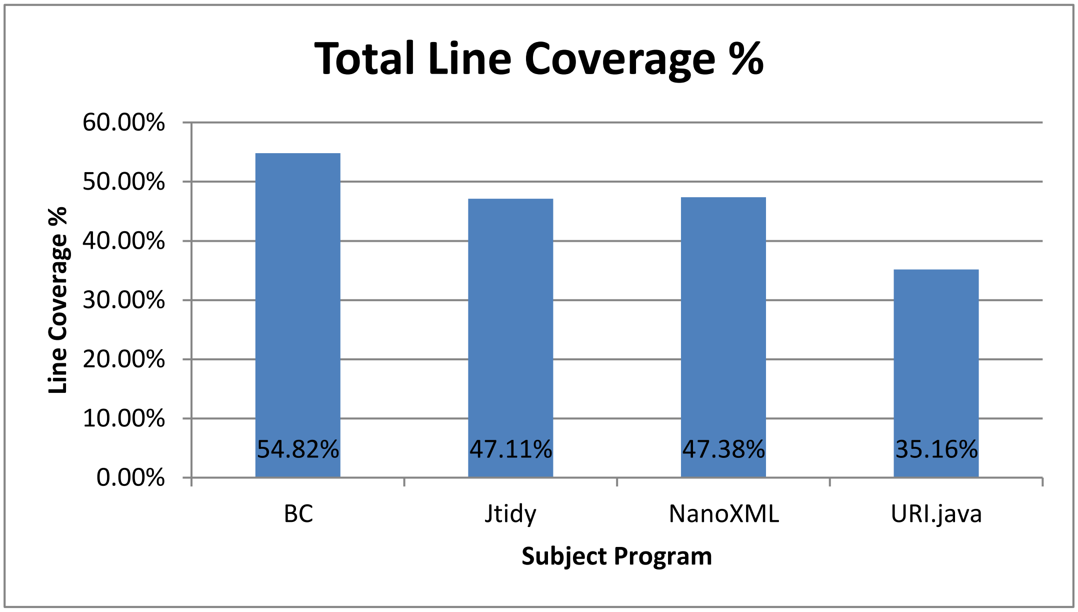
5.4.1 Overview

To begin with, we present Figure 5.2, a graph showing the line coverage percentage for each program. These percentages show the combined coverage of all four inputs (non-final, valid, invalid token, and the benchmark). The values range from just over 35% to a bit under 55%. This means that a good chunk of each program has been covered, but there remains significant functionality not covered for each program as well. URI.java likely has the least code coverage since only one constructor for the class was called, leaving any other constructors and any methods not called by the constructor untouched. We could have easily tried more constructors with more arguments, but the focus of this thesis was not to maximize the total code coverage, but rather to determine the code coverage achievable with our string inputs alone. These subject programs have some functionality that extends beyond simply testing the syntax of their inputs, so the code left uncovered is not a need for concern. This graph is intended to give a sense of how much of each program's code is covered, but it is not the end goal of this thesis to simply maximize coverage through every way available.

Table 5.1 shows all of the values for line and branch coverage for all of the subject programs. We can observe the values for all 15 combinations of the inputs. The raw data in this table provided the means for all of the analysis performed in this section of the thesis. We can observe that the number of lines of code and branches covered for the benchmark is less than that of the other three input types for three of the four subject programs. The benchmark inputs particularly struggle with NanoXML, where a total of two branches are covered compared to 92 for the next smallest value.

We do not analyze the branch coverage for the subject programs in depth because

Figure 5.2 Total line coverage percentage for all programs.



gcov and Cobertura differ in how they measure it. Additionally, in many cases, branch coverage provides less variation in its values. For instance, 10 of the 15 values for URI.java's branch coverage are 98, with 2 more being 97. While some values for line coverage may be the same or similar, particularly for URI.java or NanoXML, there is still far more variation compared to branch coverage. Since some of these values are rather close, we decided to perform our analysis on lines covered as integer values rather than as percentages, as the difference between 47.24% and 47.38% is less interesting than the difference between 702 and 704. The results for line coverage will be discussed in more detail in the rest of this section.

Table 5.1 Line and branch coverage results for all combinations of non-final, valid, invalid token, and benchmark program inputs for each of the four subject programs.

Subject Program	**Bc**	**Jtidy**	**URI.java**	**NanoXML**
Maximum Possible Line Coverage	2895	8543	1004	1486
Benchmark Line	938	3933	231	55
Invalid Token Line	1244	3319	345	462
Invalid Token + Nonfinal Line	1397	3381	346	704
Nonfinal Line	1306	3070	232	702
Valid Line	1539	3119	344	426
Valid + Invalid Token Line	1552	3335	352	462
Valid + Invalid Token + Nonfinal Line	1562	3381	353	704
Valid + Nonfinal Line	1557	3218	349	702
Invalid Token + Benchmark Line	1303	4017	348	462
Invalid Token + Nonfinal + Benchmark Line	1444	4025	348	704
Nonfinal + Benchmark Line	1389	4013	256	702
Valid + Benchmark Line	1578	4011	347	426
Valid + Invalid Token + Benchmark Line	1580	4022	353	462
Valid + Invalid Token + Nonfinal + Benchmark Line	1587	4025	353	704
Valid + Nonfinal + Benchmark Line	1586	4021	351	702
Max Possible Branch Coverage	1765	2133	305	271
Benchmark Branch	590	1139	55	2
Invalid Token Branch	782	1013	98	98
Invalid Token+ Nonfinal Branch	894	1027	98	159
Nonfinal Branch	848	944	65	159
Valid Branch	1002	984	97	92
Valid + Invalid Token Branch	1006	1021	98	98
Valid + Invalid Token + Nonfinal Branch	1006	1027	98	159
Valid + Nonfinal Branch	1006	997	98	159
Invalid Token + Benchmark Branch	810	1159	98	98
Invalid Token+ Nonfinal + Benchmark Branch	918	1160	98	159
Nonfinal + Benchmark Branch	892	1160	67	159
Valid + Benchmark Branch	1012	1160	97	92
Valid + Invalid Token+ Benchmark Branch	1012	1160	98	98
Valid + Invalid Token+ Nonfinal + Benchmark Branch	1012	1160	98	159
Valid + Nonfinal + Benchmark Branch	1012	1160	98	159

5.4.2 Unique Lines Covered for All Input Types

Figures 5.3 to 5.6 show the line coverage of the four input types. The first bar in each of these graphs displays the total number of lines covered by all four inputs combined. The other bars show the value of the line coverage for each individual input type where each bar is a stacked bar. The blue portion of the bars reveals the coverage for that input type that is common to the other three coverage types. The red portion of the bars indicates the number of lines of code covered by that input alone. Determining how many lines of code are covered uniquely for each input type is important in order to see which inputs are of the greatest value for each program. If an input type has low total coverage, then it leaves much of the program uncovered, but if it has a high total coverage but little to no unique coverage, then it is redundant with respect to the other inputs and could be removed with no loss of coverage.

We computed the unique coverage values by taking the data in Table 5.1 and applying set theory. We used the following formula to compute the unique coverage values:

```
Coverage(unique to chosen input type) =
Coverage(union of all input types) -
Coverage(union of all input types except the chosen input type)
```

Figure 5.3 shows the line coverage graph for `bc`. We observe that valid inputs have both the most total lines of coverage and the most unique lines. `bc` likely has the greatest coverage and unique coverage for valid inputs because the calculator program just wants to take in valid inputs and then perform its primary function of running calculations. `bc` cannot perform calculations with invalid input, as most of the input for non-final, invalid token, or the benchmark categories are likely to be. The

Figure 5.3 bc line coverage.

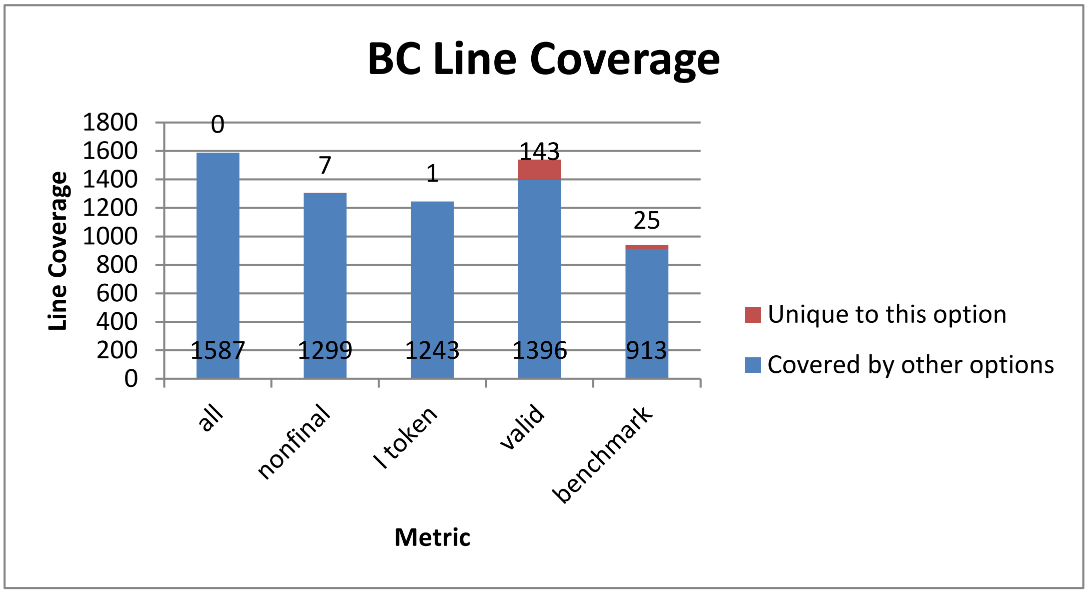


Figure 5.4 Jtidy line coverage.

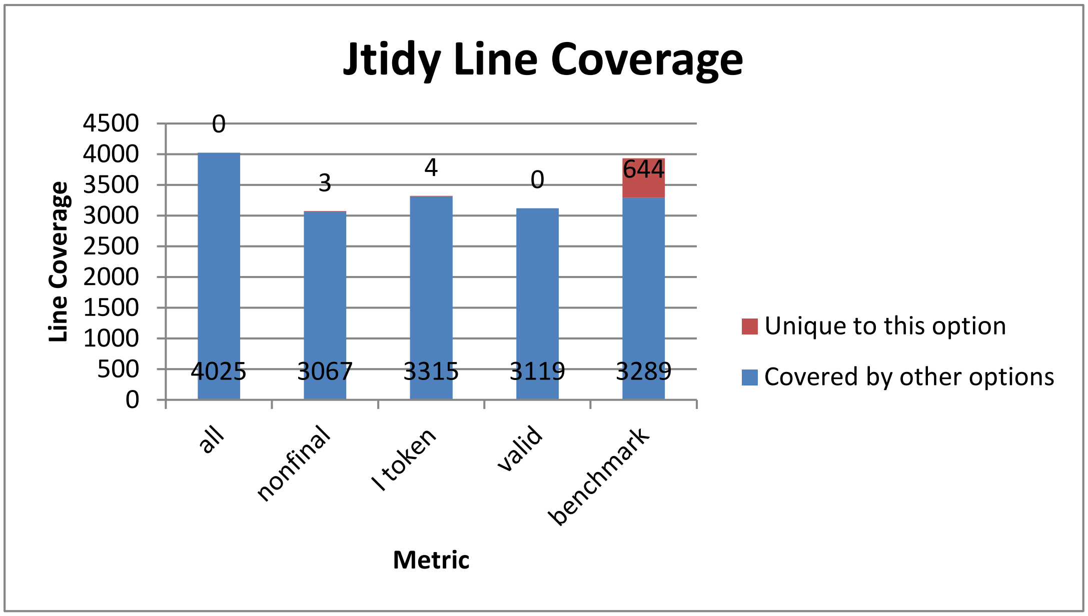


Figure 5.5 NanoXML line coverage.

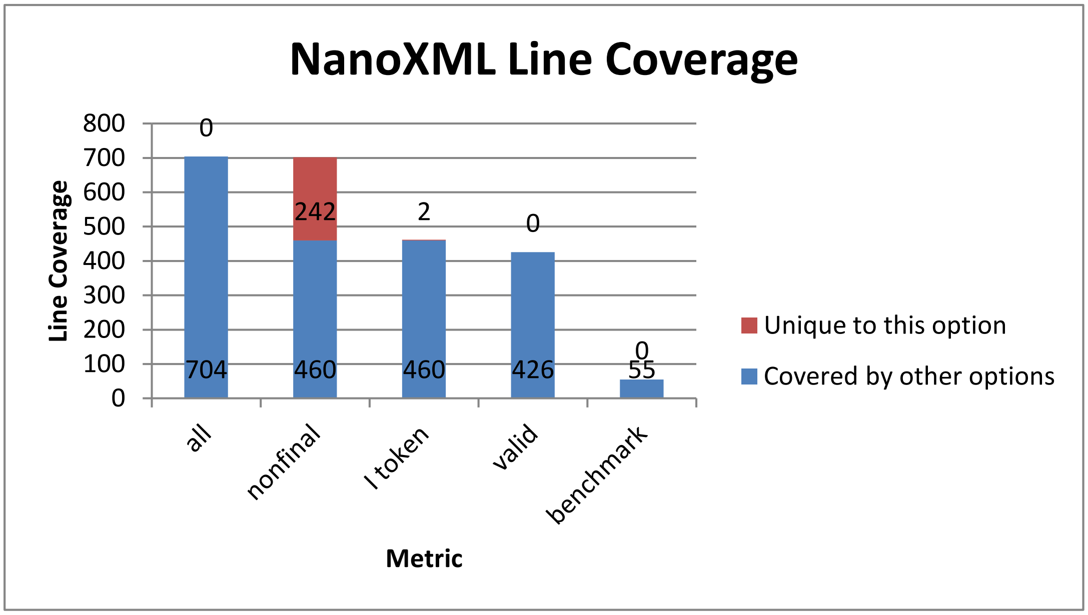
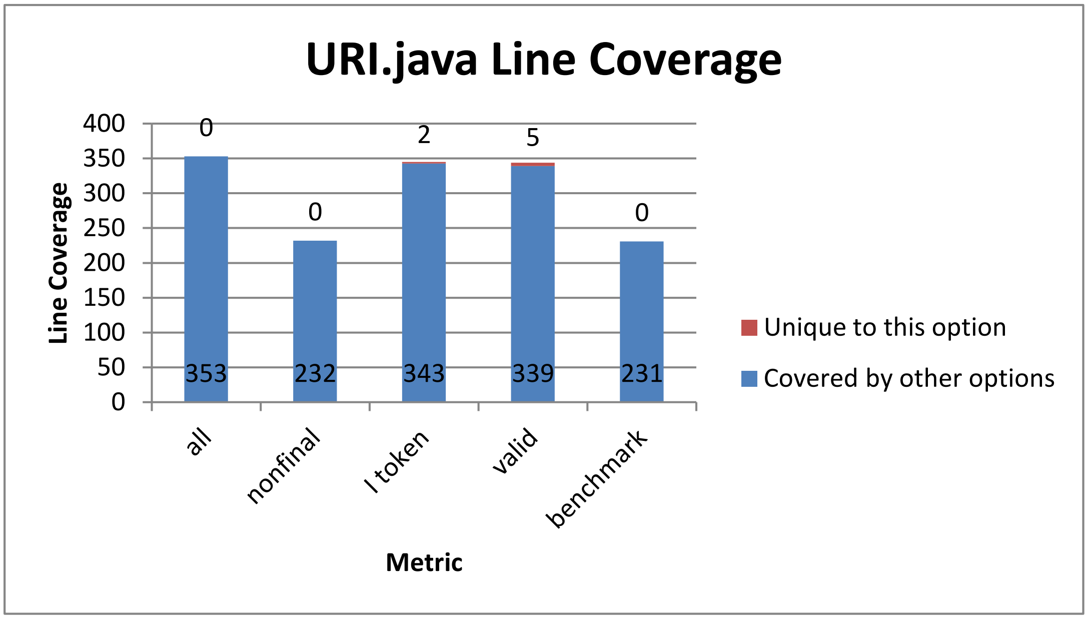


Figure 5.6 URI.java line coverage.



benchmark inputs had the least coverage but the second most unique lines covered. This could be that because the benchmark inputs might be the least likely to run `bc`'s mathematical functionality, the random distribution of the inputs could lead to testing `bc`'s order of operations checking routines in a more meaningful way than the other input types. Curiously, despite having more invalid token inputs than either of non-final or valid, invalid token has the least coverage and the least unique coverage with only one unique line of code covered.

Figure 5.4 shows the line coverage graph for Jtidy. Jtidy is the only subject program where the random inputs chosen as a benchmark achieved greater code coverage than all of the LR parse table inputs. Jtidy's benchmark not only has the most coverage, but also has the most unique lines of code covered with 644. The other three input types have three, four, and zero unique lines of code covered for non-final, invalid token, and valid inputs respectively. This makes sense when one considers the purpose of the Jtidy program. Jtidy's purpose is to fix the input given to it, so input that requires more fixing is likely to trigger more code. The randomly chosen benchmark inputs have no structure, forcing Jtidy to work extra hard to fix them.

Figure 5.5 shows the line coverage graph for NanoXML. In it, we observe that the coverage is dominated by the non-final inputs. This is what we should expect considering that we have 129 of them compared to 6 and 8 for valid and invalid token inputs respectively. Non-final inputs have 242 unique lines covered, compared to 2 for invalid token and 0 for valid and benchmark inputs. In truth, invalid token and valid inputs may have fared better than expected given how few inputs they had. The benchmark inputs covered only 55 lines of code, easily the worst relative to the other input types for all of the subject programs. With nearly 24 times the number of benchmark inputs as valid inputs resulting in less than a seventh of the coverage, it is clear that unlike Jtidy, NanoXML has no preference for random inputs to correct. This indicates just how difficult it is to find inputs that conform to the XML grammar by chance alone.

Figure 5.6 shows the line coverage for URI.java. We can see that non-final and benchmark inputs have almost exactly the same amount of total coverage. Both also have identical values of zero unique lines of code covered. The invalid token and valid inputs formed another pair with greater coverage. Invalid token inputs triggered 345 lines of code, slightly more than the 344 for valid inputs. Invalid token inputs had two unique lines of code covered compared to five for valid inputs. The inputs to URI.java had by far the fewest unique lines of code covered.

5.4.3 Unique Lines Covered for LR Parse Table Inputs

These next graphs are much the same as those discussed previously, the difference being that the benchmark inputs are removed from the comparison. These graphs are included so as to observe the unique lines covered between the different LR parse table strategies without the benchmark inputs getting in the way. We have included graphs for all subject programs, but the focus is on the graph for Jtidy, since the removal of the benchmark coverage has the greatest effect there.

The graph in Figure 5.7 shows the line coverage for the `bc` subject program without the benchmark inputs. This graph is largely the same as the graph with the benchmark inputs, merely with a few more unique lines of coverage spread out. The slight increase in coverage for the valid inputs is a bit curious as one would expect next to none of the benchmark inputs to have been valid at all (and thus unlikely to

Figure 5.7 bc line coverage without benchmark.

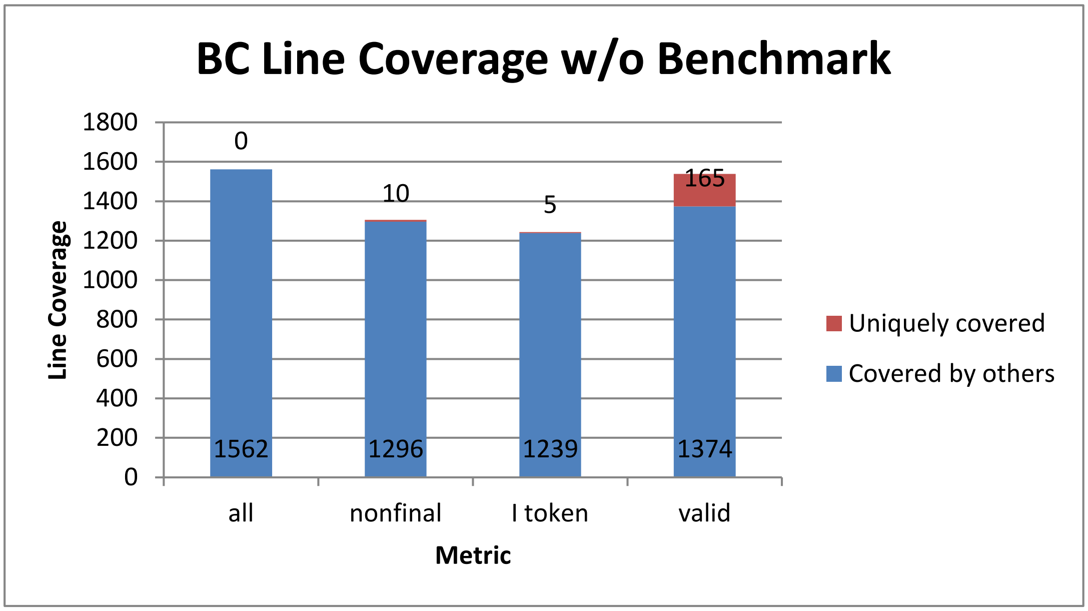


Figure 5.8 Jtidy line coverage without benchmark.

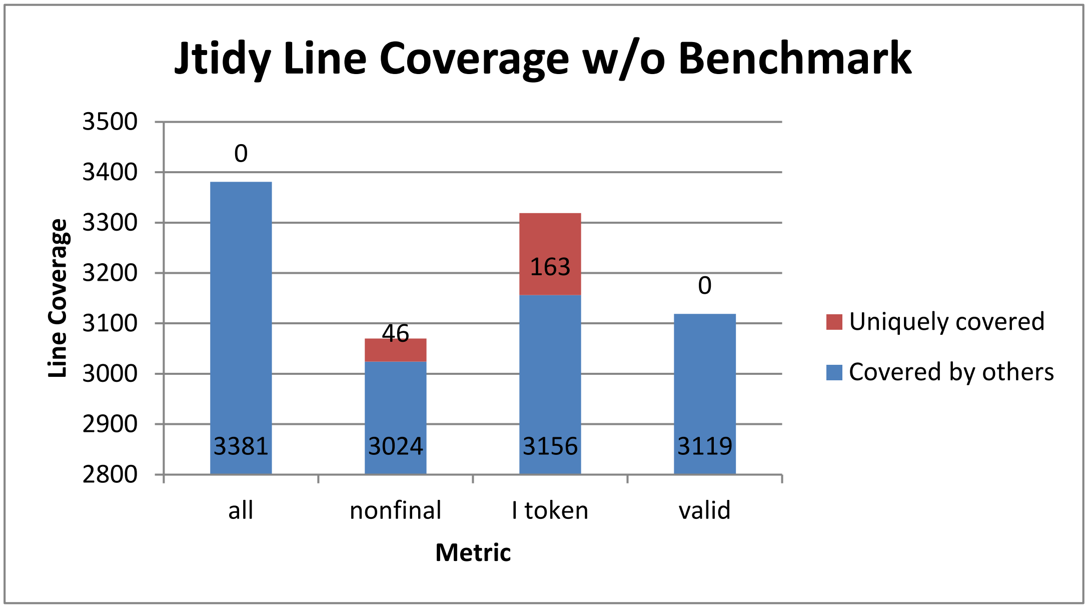


Figure 5.9 NanoXML line coverage without benchmark.

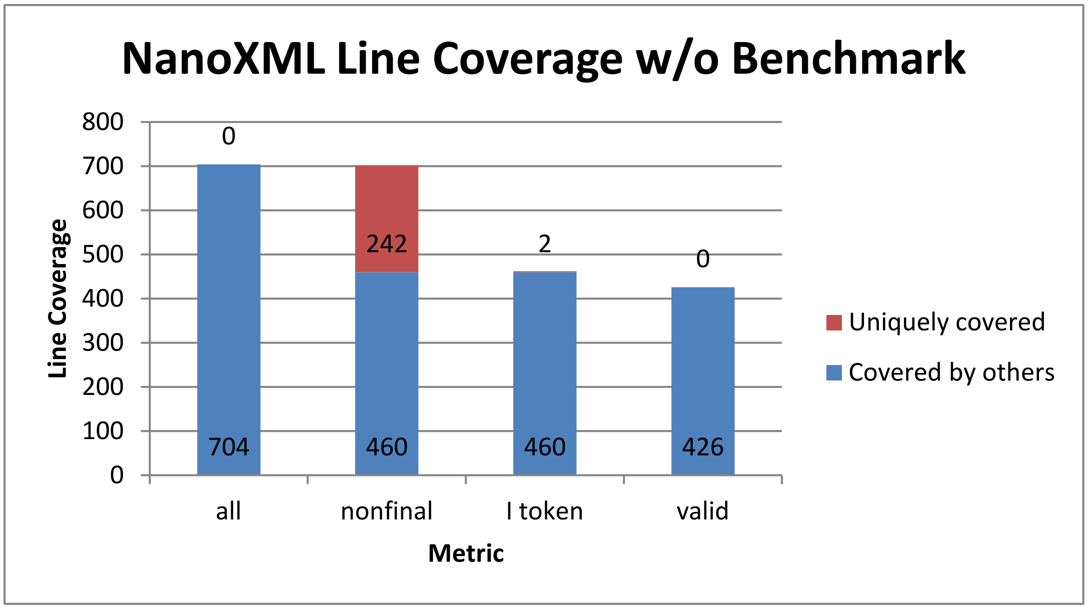
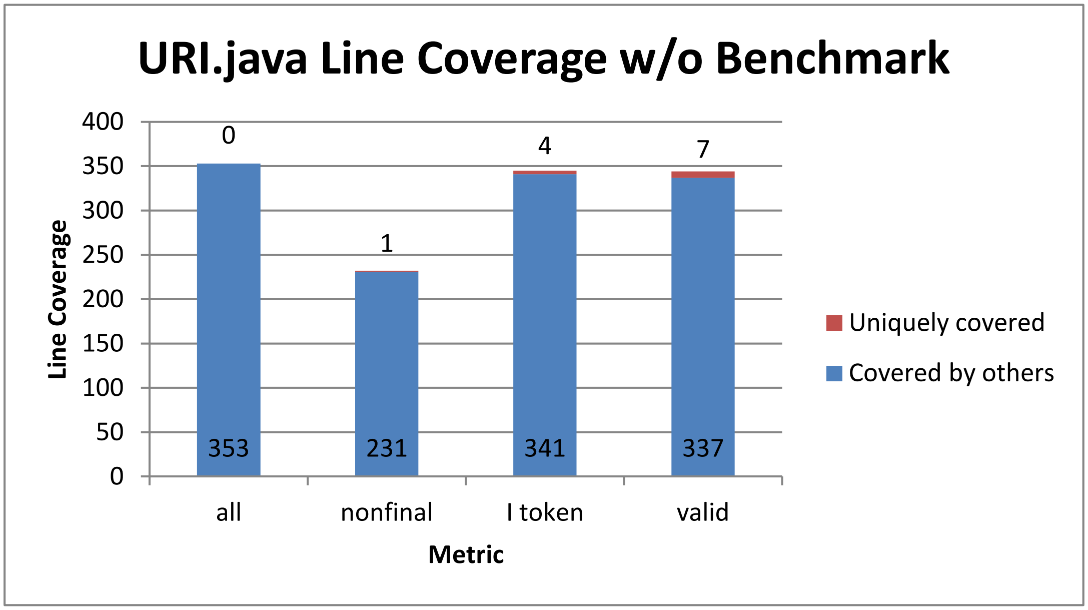


Figure 5.10 URI.java line coverage without benchmark.



stimulate code otherwise unique to valid inputs). The graph in Figure 5.9 shows the line coverage for the NanoXML subject program without the benchmark inputs. The number of unique lines of code in each category has not changed at all, presumably because of the ineffectiveness of the benchmark inputs of triggering code. The graph in Figure 5.10 shows the line coverage for the URI.java subject program without the benchmark inputs. This graph shows very slight improvements across all categories with the removal of the benchmark inputs, but nothing significant.

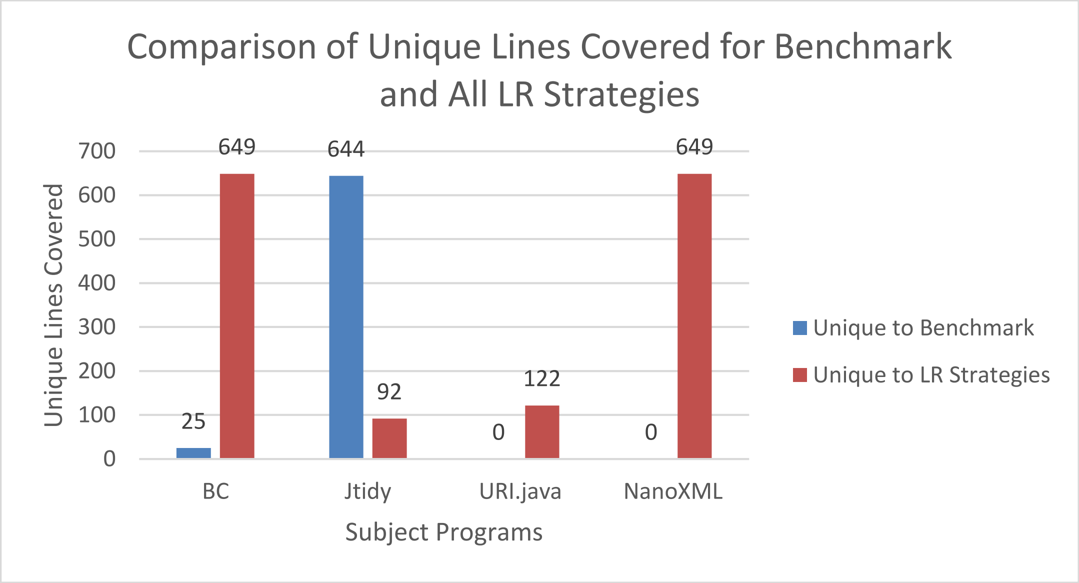
The graph in Figure 5.8 shows the line coverage for the Jtidy subject program without the benchmark inputs. This graph has the most significant change from the previous version because the benchmark inputs triggered so much code for this program. We can see from this graph that the invalid token inputs have both the greatest total code coverage and by far the most unique code coverage. Some of this may simply be due to the large number of invalid token strings that we were able to generate relative to the other categories, but much of it can likely be attributed to Jtidy's need to correct errors. These 163 unique lines of code covered by the invalid token inputs were all but hidden by the benchmark input coverage previously. The non-final inputs have the least coverage, but they also provide 46 unique lines of code covered, all but three of which had been overwhelmed by the benchmark input coverage before. This may imply that these inputs have tripped some code in Jtidy intended for completing outputs separate from the code in the other categories. The valid inputs for Jtidy cover 3119 lines of code, but none of this is unique. Presumably the same code that Jtidy uses to affirm the valid inputs is triggered for the invalid token inputs, just that the invalid token inputs also trigger the code then required to fix more of the inputs.

5.4.4 Unique Lines Covered for LR Parse Table Inputs Compared to Benchmark Inputs

The benchmark inputs are equal in number to the combined non-final, invalid token, and valid inputs for each subject grammar. However, every benchmark input is the maximum input length, whereas only the longest of the LR parse table inputs are that long. This means that the benchmark inputs have at least an equal, if not greater than equal, chance at inducing more code coverage by volume. As such, Figure 5.11 compares the unique lines covered for the combined LR strategies against the unique lines for the benchmark. The blue bars show the lines covered by the benchmark but not the LR strategies, and the red bars show the lines covered by the LR strategies but not the benchmark. In the previous graphs in this section, many of the values of unique lines covered for the LR parse table strategies were reduced because if one strategy covered a line, then it could not be counted for the unique coverage of another strategy. This new graph removes that obstacle. The fact that two values in the graph are exactly 649 and three of the values are in the 640s is just coincidence.

Figure 5.11 shows 25 unique lines of code covered by the benchmark inputs for the `bc` program compared to 649 unique lines covered by the combined LR parse table inputs. This indicates that the `bc` program has nearly 26 times more code covered only by the LR parse table inputs than the `bc` benchmark inputs of equal volume. The Jtidy program has 644 unique lines of code covered by the benchmark inputs, which is easily the most for all of the subject programs. While small by comparison, we still see a healthy 92 lines of code covered by the LR parse table inputs that the benchmark inputs have left untouched. This would indicate that even when a program which prefers input consisting of many random errors is used, the LR parse table inputs

Figure 5.11 Lines covered by the benchmark but not the LR strategies versus lines covered by the combined LR strategies but not the benchmark.



still contribute a significant portion of code coverage that would not otherwise be possible. For URI.java and NanoXML, the LR parse table inputs achieve 122 and 649 unique lines of code covered compared to zero for the benchmark. This shows the great effectiveness of the LR parse table strategies for these programs. Therefore, for 3 out of the 4 programs, the randomly chosen inputs add little to no coverage not provided by the LR parse table strategies, and when the subject program does prefer inputs that are heavily garbled and full of errors, the LR parse table strategies still have significant complementary value.

5.5 Test Results

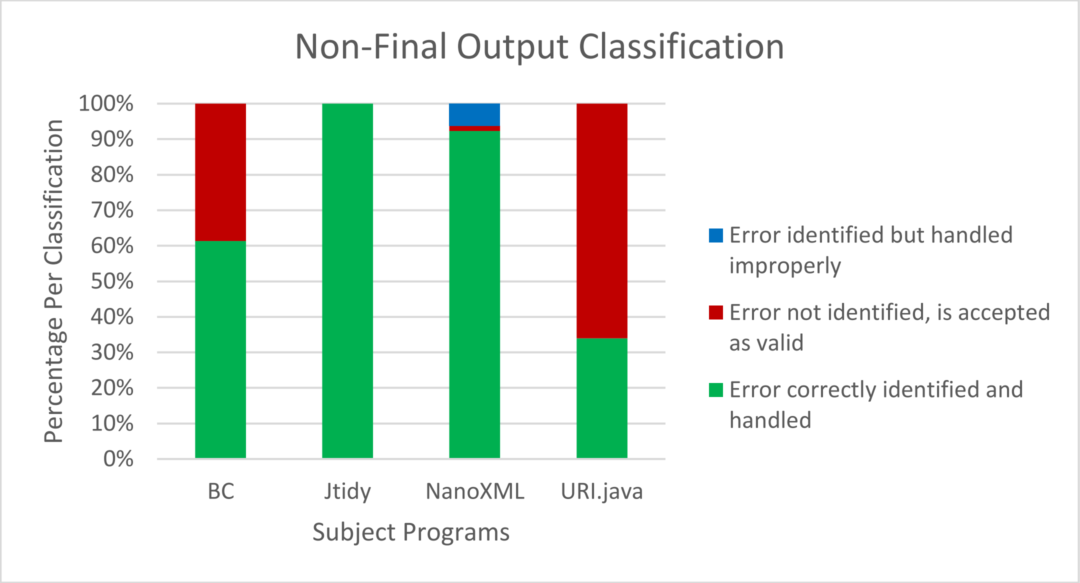
The point of software testing is not merely to achieve code coverage, but rather to ensure that a program is providing correct outputs. We observed if our valid inputs were accepted by our subject programs and if our invalid inputs (both invalid token and non-final) were correctly identified by the programs and handled in a controlled manner. If valid input is rejected or invalid input is accepted, then there must be some difference between the way the grammar is set up and the way the subject program tests if its inputs conform to the grammar. Additionally, if a subject program cannot handle an invalid input (or a valid one for that matter) and simply crashes rather than handling it in a controlled manner, then that is something that software being tested would need to fix too. We evaluated the percentage of inputs in each category for each subject program that have been handled correctly or incorrectly based on the outputs triggered.

5.5.1 Non-Final Output Classification

Figure 5.12 shows how we classified the non-final outputs. There is a separate bar for each subject program, all out of 100% so as to keep the scale the same. We classified non-final outputs into three categories: errors that had been identified by the program but handled improperly (such as causing the program to crash), errors not identified that were accepted as valid, and errors identified correctly and handled in a controlled manner. There was significant variation between subject programs so we shall now discuss each in turn.

The `bc` subject program correctly identified approximately 60% of its non-final inputs

Figure 5.12 Classification of non-final outputs.



as errors, with 40% erroneously accepted as valid, and none causing the program to crash. Most of the error messages that `bc` provided when identifying erroneous inputs were "parse errors," which is an acceptable classification. While it is good that none of the inputs crashed the `bc` program, having that many invalid inputs accepted as valid is cause for concern. However, the reason for this may simply be because `bc` permits the user to exit partway through making a calculation. A user using `bc` as an interactive mathematical shell may wish to quit partway through entering an input for a sophisticated calculation if they discover that they have made an error or simply do not require the calculation. A message informing the user that entering `5 + 7 -` is not a complete mathematical statement may not be necessary. The BC grammar page says that the "exit status for error conditions has been left unspecified" [27]. Since we call the "quit" function immediately after every input, this might cause some of the non-final outputs to lack error messages.

The Jtidy program printed messages indicating incorrect input consistently. In general, Jtidy seemed to find something wrong with almost *every* input, regardless of if it was invalid or valid. While this has its drawbacks, it means that all of Jtidy's non-final inputs were correctly addressed as being invalid. Jtidy attempts to fix invalid inputs provided to it, so if it sees that something is missing at the end of an input it will add it.

NanoXML identified and handled more than 90% of its non-final inputs correctly. However, it also had 8 instances where the program crashed. In these instances, NanoXML threw `NullPointerException`s. These all occur at the same point in the NanoXML code, suggesting a single cause. In the code, a function intended to scan the data entered fails to return an object to a method intended to parse the input. This object, which is null, is cast to the type "XMLElement," which is then dereferenced by a call to a write method. The write method, which the comments indicate expects a non-null XML Element to write, checks the name of the XML Element (named "xml") in an `if` statement with `if (xml.getName() == null)`. Since the XML Element xml is null, calling the getName() function on xml will result in a `NullPointerException` because there is no object xml to call the function on. This exception is returned back up to the main method, which has no "try - catch" in place to catch the exception, and so the program crashes. The `NullPointerException`s were thrown by the inputs numbered 9, 11, 12, 14, 15, 19, 97, and 117 and are shown in Figure 5.13. We analyzed these inputs compared to the other non-final NanoXML inputs but could not detect any clear pattern. Regardless, it is unacceptable for a program like NanoXML to crash on this many different inputs. It is surprising that a program this old would not have had this fault corrected yet, given that the last update to NanoXML occurred 13 years ago [12].

Figure 5.13 NanoXML `NullPointerException` Inputs

```
9: <?xml version = '1.1' ?>
11: <?xml version = '1.1' ?><!--Comment - abc432&*(-->
12: <?xml version = '1.1' ?><!--Comment-->
14: <?xml version = '1.1' ?><?PI abc123&*(?>
15: <?xml version = '1.1' ?><?PI?>
19: <?xml version = '1.1' encoding = "EncName"
97: <?xml version = '1.1' encoding = "EncName" standalone = "yes"
 ?>  <!DOCTYPE Name >
117: <?xml version = '1.1' encoding = 'EncName'
```

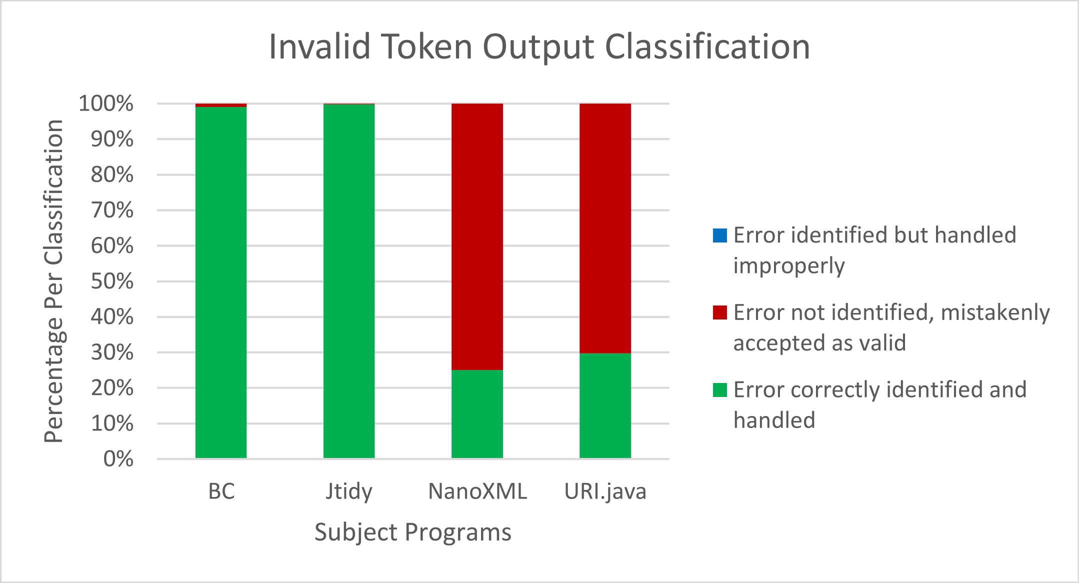
URI.java classified slightly over 30% of its non-final inputs correctly and failed to identify the rest as erroneous though no inputs caused the program to crash. This is not an acceptable number of invalid inputs to have accepted. While the URI subject program did make reference to being slightly more lenient in some areas than the grammar which it was designed to test, having this many non-final inputs classified incorrectly could be improved by simply flipping a coin. For those that were rejected, a URI syntax exception was thrown, which is an acceptable means of handling the erroneous inputs caught.

5.5.2 Invalid Token Output Classification

Now we shall discuss the invalid token outputs, classified in Figure 5.14. The classification scheme is much the same as it was for the non-final outputs.

The `bc` and Jtidy subject programs performed brilliantly on the invalid token inputs, with nearly all of their outputs indicating a correctly found and handled error. `bc` threw parse errors for most of the outputs, as it did for the non-final ones. `bc` likely performed so well because it is a calculator, and checking that all of the components

Figure 5.14 Classification of invalid token outputs.



follow a correct order of operations in important for calculators. Since invalid token inputs are essentially changing one step of the order of operations to be incorrect, `bc` is likely to find these errors. Jtidy located problems for all of its inputs because, as previously mentioned, it finds something wrong with almost every input given to it regardless of category. The main difference for Jtidy on invalid inputs, compared to non-final or valid ones, is that it explicitly mentions that about 100 of the inputs have an "error," whereas normally it just provides one or more "warnings." When Jtidy finds an issue to warn the user about, it corrects the input to be in line with the warning. However, when Jtidy encounters an error, no correction is made and the program simply provides any warning and error messages to the user and then exits.

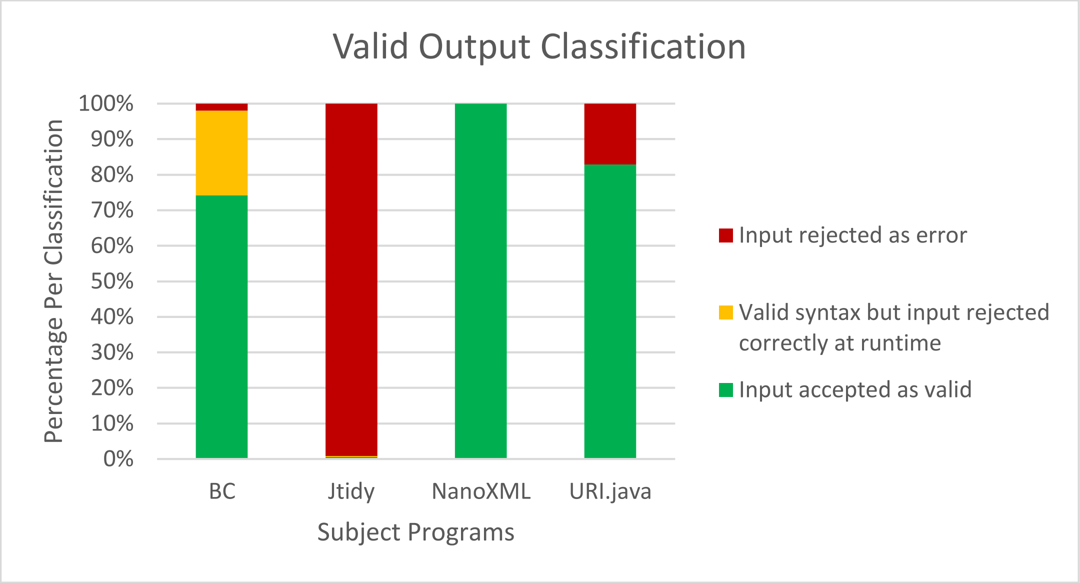
NanoXML and URI.java each failed to identify a significant number of their invalid token inputs correctly, finding between 20% and 30% in both cases. NanoXML only had 8 inputs, so the fact that only two of these threw exceptions is all the more concerning. Considering how difficult it was to obtain these strings for NanoXML, it is a bit disappointing that these errors were not located. It seems that NanoXML can generally tell when an input ends early (non-final inputs), but it is not good at picking out an error in a completed input. In that sense, NanoXML is the opposite of the `bc` subject program. URI.java missed many errors as well, with a similar percentage of effectiveness as it had for non-final inputs. This may suggest that the reason for URI.java's ineffectiveness is the same in both cases: that the subject program differs in some areas from the grammar in terms of how it evaluates inputs, as the documentation suggests. On the plus side, no invalid token inputs caused any of the subject programs to crash.

5.5.3 Valid Output Classification

Now we shall discuss the classification of valid output, as shown in Figure 5.15. We broke the output of the supplied valid inputs down into three categories: the input is accepted as valid, the input syntax is accepted but the input is rejected at runtime due to a logic error, and the input is rejected as being invalid by the subject program. Note that outputs in the second category are still considered to be handled well by the program.

The `bc` subject program handled most of its valid input correctly. While there was one parse error, all of the rest of the outputs suggested that `bc` had handled them well. `bc` had 26 runtime errors, mostly either due to a function not being defined or because a calculation tried to divide by zero. Our inputs could not take these issues into account, so `bc` locating them is the correct and expected result. The rest of the

Figure 5.15 Classification of valid outputs.



inputs were all considered valid.

Jtidy performed the best out of the four subject programs at locating problems in non-final and invalid token inputs. Unfortunately, Jtidy continued to find many problems with what should have been considered valid inputs. Whereas programs like NanoXML or URI.java might be slightly more lenient generally than the grammar that they are based on, Jtidy appears to be more critical than the HTML grammar. Jtidy did not classify any of the valid inputs as errors necessarily, but it did provide many warnings. Since these same warnings were classified as acceptable evidence of locating invalid input, it is only fair that they be classified as evidence of rejecting valid input now. Nonetheless, it must be noted that Jtidy did not refuse any valid input as being errors that could not be handled. It is curious that Jtidy with its natural fault-finding ability would have so many issues with the HTML outputs, yet the valid inputs triggered no unique code coverage. This must mean that any problems found with the valid inputs must also have been found with the non-final or invalid token inputs too.

NanoXML accepted all of its valid inputs as valid according to the outputs. This is appears encouraging, but considering how many of the invalid token inputs were also accepted, this may simply indicate that NanoXML does not test its complete (rather than non-final) inputs thoroughly. We should also note that there were only 6 valid inputs for NanoXML, so it may be too early to draw conclusions.

URI.java accepted more than 80% of its valid inputs as valid. The other inputs were rejected as invalid. URI.java threw URI Syntax Exceptions for the inputs which it rejected as invalid. Analysis of how the subject program alters what it looks for compared to the original RFC indicates that at least some of these inputs rejected as errors are not rejected solely because of the outlined differences in evaluation. Many inputs are rejected due to something being wrong with the schemes of the URIs, and there is no mention of any changes to the schemes in the list of differences. That the program accepts about 70% of the non-final and invalid token inputs, yet accepts only 10% more of the valid inputs, is too small of a difference to be comfortable with. That small of a margin could be by chance.

Overall, we see that the outputs of the subject programs suggest that our programs all handled their inputs in significantly different ways. This means that choosing all of these programs to test on was a good idea since it gives us a better grasp of what the possible outcomes are than if we had either chosen fewer programs or chosen programs more similar to each other, leading to inaccurate generalizations. The downside of this variation is that it makes it difficult to pick out a general trend across all programs that can be used for predictive purposes.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

An insufficient amount of work has been done over the years for testing grammar-based software with invalid string inputs. In this thesis, we addressed this need by generating LR parse tables for context-free grammars and then finding strings that would achieve coverage of these tables. We created strings of three types: those that reach a non-final state in the LR parse table and then stop, those that start in the start state, go through a particular non-final state, and then continue on to completion to the final state with valid input, and strings that go from start to finish but have the first token after the non-final state exchanged for a terminal that is invalid at that state. We used a depth-first iterative deepening algorithm with heuristic stopping conditions to generate these strings. We also generated strings to use for comparison where each token was chosen randomly. We converted all of the strings to bytes suitable as input to subject programs.

We applied the inputs to the programs and measured the code coverage to discern the inputs' effectiveness in testing the subject programs. We also evaluated the outputs of the subject programs to see if the programs rejected invalid input in a controlled manner and accepted valid input. The experimental data suggests that the LR parse table generated inputs generally achieved greater code coverage and unique code coverage compared to the benchmark inputs. The number of inputs that were correctly classified as valid or invalid was highly program dependent.

More specifically, this thesis makes the following conclusions. Firstly, that due to the state space explosion problem of the LR parse table where an infinite number of stacks and routes through the table may be possible, it is not always feasible to obtain strings to reach a state or reach a state and then continue to completion. Secondly, that for the majority of subject programs that this thesis analyzed, the LR parse table generated strings achieved greater code coverage than the benchmark strategy of choosing each element in the string randomly. Third, that even when the benchmark achieved higher code coverage, the LR parse table inputs achieved coverage of a significant portion of code that would be left untouched by the random testing strategy. Finally, upon analyzing the non-final, invalid token, and valid outputs for each program, we conclude that the number of inputs correctly classified as valid or invalid in a controlled manner depended largely on the program under test. Some programs crashed on certain inputs, some programs rejected almost all inputs as invalid, and some differed too much from the grammar that they were based on to be accurately represented by the LR parse table from which their inputs were generated. The programs differed greatly in terms of how they handled their inputs, and so provided a balanced representation of issues to be encountered in real-world grammar-based testing. However, this variation also prevents us from declaring any general trends for the number of inputs to be classified correctly by grammar-based software.

6.2 Future Work

This thesis has provided an analysis of the effectiveness of using valid and invalid strings generated from LR parse tables for the purpose of testing grammar-based software. This thesis has opened up further areas of research which can build on the work we performed. Work in these additional areas can improve upon, and clarify, the results of this thesis, and so we shall discuss some of these opportunities for future work on LR parse table based testing now.

One area of future work would be to expand upon the invalid token string generation portion. To do this, one could try creating a string for *every* token for which there is no transition when choosing invalid tokens at a state, rather than only one randomly chosen token. This would generate an enormous number of strings, too many for efficient testing. However, it might be possible to look for trends in all of these strings and to partition the strings into equivalence classes so as to maximize code coverage for a reasonable number of inputs. Currently, we choose an invalid token randomly amongst the possible terminals, but perhaps there is some other means of choosing invalid tokens that would be more effective. For example, perhaps ensuring that every token gets used at least once as an invalid token will test out more code than if the distribution is random? Perhaps using an invalid token that is valid in a preceding or succeeding state would achieve higher code coverage because the subject program might be primed to deal with tokens where the order is "off by one?" There are many such possibilities that can be explored.

Another area of further research would be to create smarter algorithms for searching through the LR parse tables. Given how few valid and invalid token inputs we were able to generate for XML, and our struggles in reaching a large number of the URL grammar's non-final states, improved algorithms would be of great value in obtaining more inputs to use for testing the subject programs. For starters, perhaps an algorithm could alter its values of RPS and maximum depth while performing a search based on how difficult it determines its current area of the parse table is to travel through. Another possibility is to have the algorithm check if all of the goto transitions for a state have been used up after all of the shifts and reductions had been taken, and then guide the search so as to give the best chance of triggering these gotos. This was considered, but not studied in depth, due to time constraints in our work.

Additional research could be done by performing a deeper analysis of the error messages provided by invalid (or valid) inputs to the subject programs. Research could be done to determine if errors are thrown or not thrown due to the program code not completely correlating with the intention of the grammar, or if the subject program did intend to test that aspect of the grammar but merely failed. Code coverage from the different input types and other metrics could also be used to determine how well programs correlate with the grammars they are supposed to be based on.

Obtaining more programs could be useful for spotting correlations. These correlations could be between the number of strings generated in each category, the quantity of productions in the grammar, the number, and ratios of, shift, reduce, and goto transitions, and the numbers for any other metrics thought potentially useful. We did not perform such analysis ourselves because statistical calculations with only four data points are unreliable. With more subject programs, a researcher could also learn which of non-final, valid, invalid token, or the benchmark strategies performs best for different program types, as we could only make generalizations from the limited data available. It could also be worth trying collections of multiple subject programs where each collection of programs is based on a single grammar. This could eliminate confusion as to when a particular result is determined by the program and when it is determined by the grammar.

Some analysis could also be performed on the lengths of the strings used to travel through the LR parse table. Our depth-first iterative deepening algorithm generated very efficient strings to reach its states. If these strings took longer routes through the table to reach or go through the same states, would there be a noticeable increase in code coverage? Most of the strings generated for each grammar did not extend to the maximum depth aside from those of the benchmark, so it could also be worth analyzing these existing strings to check for a correlation between their length and the coverage they achieve. Given that the benchmark strategy, where every string was the maximum length, fared poorly compared to the shorter LR parse table strategies' strings for three of the four programs, length alone must not be the only criteria determining code coverage.

Future work could also include performing further analysis on our benchmark strategy. A researcher could try a random tester where inputs are chosen completely randomly, regardless of the terminals available for a grammar. A researcher could also try randomly distributed collections of valid terminals rather than randomly choosing each terminal separately. In this situation, all of the elements for a valid string would be provided, just that each collection of some size (say, every 3 terminals for example) would have its location exchanged for that of another collection of equal size. Alternatively, one could keep the collections in the same order, but rearrange the terminals within each collection. A researcher could also consider analyzing the strings generated from the benchmark (the current one or the other versions mentioned) and observing how many transitions into the LR parse table the strings get. This could also lead to a study of the probability of getting from the start state to another given state (such as the final state) purely by chance. (It would likely be a small probability for getting all the way through, particularly for XML).

Overall, this thesis has made a significant contribution to the field of grammar-based software testing. However, while much has been learned, there is considerably more that can be explored in this research area.

Appendix A

Example Grammar LR Parse Table

```
-------- ACTION_TABLE --------

From state #0

[term 4:SHIFT(to state 3)] [term 9:SHIFT(to state 5)]

[term 11:SHIFT(to state 2)]

From state #1

[term 0:REDUCE(with prod 2)] [term 4:REDUCE(with prod 2)]

[term 9:REDUCE(with prod 2)] [term 11:REDUCE(with prod 2)]

From state #2
```

```
[term 2:REDUCE(with prod 11)] [term 3:REDUCE(with prod 11)]
[term 4:REDUCE(with prod 11)] [term 5:REDUCE(with prod 11)]
[term 6:REDUCE(with prod 11)] [term 7:REDUCE(with prod 11)]
[term 10:REDUCE(with prod 11)]
From state #3
[term 4:SHIFT(to state 3)] [term 9:SHIFT(to state 5)]
[term 11:SHIFT(to state 2)]
From state #4
[term 0:SHIFT(to state 21)] [term 4:SHIFT(to state 3)]
[term 9:SHIFT(to state 5)] [term 11:SHIFT(to state 2)]
From state #5
[term 4:SHIFT(to state 3)] [term 9:SHIFT(to state 5)]
[term 11:SHIFT(to state 2)]
From state #6
[term 2:SHIFT(to state 9)] [term 3:SHIFT(to state 8)]
```

```
[term 4:SHIFT(to state 10)] [term 5:SHIFT(to state 12)]

[term 6:SHIFT(to state 11)] [term 7:SHIFT(to state 7)]

From state #7

[term 4:SHIFT(to state 3)] [term 9:SHIFT(to state 5)]

[term 11:SHIFT(to state 2)]

From state #8

[term 4:SHIFT(to state 3)] [term 9:SHIFT(to state 5)]

[term 11:SHIFT(to state 2)]

From state #9

[term 0:REDUCE(with prod 3)] [term 4:REDUCE(with prod 3)]

[term 9:REDUCE(with prod 3)] [term 11:REDUCE(with prod 3)]

From state #10

[term 4:SHIFT(to state 3)] [term 9:SHIFT(to state 5)]

[term 11:SHIFT(to state 2)]

From state #11
```

```
[term 4:SHIFT(to state 3)] [term 9:SHIFT(to state 5)]

[term 11:SHIFT(to state 2)]

From state #12

[term 4:SHIFT(to state 3)] [term 9:SHIFT(to state 5)]

[term 11:SHIFT(to state 2)]

From state #13

[term 2:REDUCE(with prod 6)] [term 3:REDUCE(with prod 6)]

[term 4:REDUCE(with prod 6)] [term 5:REDUCE(with prod 6)]

[term 6:REDUCE(with prod 6)] [term 7:REDUCE(with prod 6)]

[term 10:REDUCE(with prod 6)]

From state #14

[term 2:REDUCE(with prod 7)] [term 3:REDUCE(with prod 7)]

[term 4:REDUCE(with prod 7)] [term 5:REDUCE(with prod 7)]

[term 6:REDUCE(with prod 7)] [term 7:REDUCE(with prod 7)]

[term 10:REDUCE(with prod 7)]
```

```
From state #15

[term 2:REDUCE(with prod 5)] [term 3:REDUCE(with prod 5)]

[term 4:REDUCE(with prod 5)] [term 5:SHIFT(to state 12)]

[term 6:SHIFT(to state 11)] [term 7:SHIFT(to state 7)]

[term 10:REDUCE(with prod 5)]

From state #16

[term 2:REDUCE(with prod 4)] [term 3:REDUCE(with prod 4)]

[term 4:REDUCE(with prod 4)] [term 5:SHIFT(to state 12)]

[term 6:SHIFT(to state 11)] [term 7:SHIFT(to state 7)]

[term 10:REDUCE(with prod 4)]

From state #17

[term 2:REDUCE(with prod 8)] [term 3:REDUCE(with prod 8)]

[term 4:REDUCE(with prod 8)] [term 5:REDUCE(with prod 8)]

[term 6:REDUCE(with prod 8)] [term 7:REDUCE(with prod 8)]

[term 10:REDUCE(with prod 8)]
```

```
From state #18

[term 3:SHIFT(to state 8)] [term 4:SHIFT(to state 10)]

[term 5:SHIFT(to state 12)] [term 6:SHIFT(to state 11)]

[term 7:SHIFT(to state 7)] [term 10:SHIFT(to state 19)]

From state #19

[term 2:REDUCE(with prod 10)] [term 3:REDUCE(with prod 10)]

[term 4:REDUCE(with prod 10)] [term 5:REDUCE(with prod 10)]

[term 6:REDUCE(with prod 10)] [term 7:REDUCE(with prod 10)]

[term 10:REDUCE(with prod 10)]

From state #20

[term 0:REDUCE(with prod 0)] [term 4:REDUCE(with prod 0)]

[term 9:REDUCE(with prod 0)] [term 11:REDUCE(with prod 0)]

From state #21

[term 0:REDUCE(with prod 1)]

From state #22
```

```
[term 2:REDUCE(with prod 9)] [term 3:REDUCE(with prod 9)]
[term 4:REDUCE(with prod 9)] [term 5:REDUCE(with prod 9)]
[term 6:REDUCE(with prod 9)] [term 7:REDUCE(with prod 9)]
[term 10:REDUCE(with prod 9)]
------------------------------
-------- REDUCE_TABLE --------
From state #0
[non term 1->state 4] [non term 2->state 1] [non term 3->state 6]
From state #1
From state #2
From state #3
[non term 3->state 22]
From state #4
[non term 2->state 20] [non term 3->state 6]
From state #5
```

```
[non term 3->state 18]

From state #6

From state #7

[non term 3->state 17]

From state #8

[non term 3->state 16]

From state #9

From state #10

[non term 3->state 15]

From state #11

[non term 3->state 14]

From state #12

[non term 3->state 13]

From state #13

From state #14
From state #15

From state #16

From state #17

From state #18

From state #19

From state #20

From state #21

From state #22

-----------------------------
```

Appendix B

Grammar Dump for Example Grammar

```
===== Terminals =====

[0]EOF

[1]error

[2]SEMI

[3]PLUS

[4]MINUS

[5]TIMES
```

```
[6]DIVIDE
[7]MOD
[8]UMINUS
[9]LPAREN
[10]RPAREN
[11]NUMBER
===== Non terminals =====
[0]$START
[1]expr_list
[2]expr_part
[3]expr
[4]term
[5]factor
===== Productions =====
[0] expr_list ::= expr_list expr_part
```

```
[1] $START ::= expr_list EOF

[2] expr_list ::= expr_part

[3] expr_part ::= expr SEMI

[4] expr ::= expr PLUS expr

[5] expr ::= expr MINUS expr

[6] expr ::= expr TIMES expr

[7] expr ::= expr DIVIDE expr

[8] expr ::= expr MOD expr

[9] expr ::= MINUS expr

[10] expr ::= LPAREN expr RPAREN

[11] expr ::= NUMBER
```

Appendix C

CUP Copyright Notice, License and Disclaimer

The authors and their employers disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall the authors or their employers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson: Addison Wesley, Boston, second edition, 2007.

[2] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[3] James H. Andrews, Felix Chun Hang Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, 2007.

[4] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (uri): Generic syntax, 1998.

[5] M. Beyene and J.H. Andrews. Generating string test data for code coverage. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 270–279, April 2012.

[6] Alexandre Blondin Massé, Sébastien Gaboury, Sylvain Hallé, and Michaël Larouche. Solving equations on words with morphisms and antimorphisms. In Adrian-Horia Dediu, Carlos Martin-Vide, José-Luis Sierra-Rodriguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, volume 8370 of *Lecture Notes in Computer Science*, pages 186–197. Springer International Publishing, 2014.

[7] Fabian Büttner and Jordi Cabot. Lightweight string reasoning in model finding. *Software and Systems Modeling*, 14(1):413–427, 2015.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] P.K. Chittimalli and M.J. Harrold. Recomputing coverage information to assist regression testing. *Software Engineering, IEEE Transactions on*, 35(4):452–469, July 2009.

[10] L.A. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, SE-2(3):215–222, Sept 1976.

[11] Devin Cook and Multiple Contributors. Gold parsing system: Multi-programming language, parser, August 2012.

[12] Marc De Scheemaecker. Nanoxml 2.2.1, April 2002.

[13] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, April 1994.

[14] R. Feldt and S. Poulding. Finding test data with specific properties via meta-heuristic search. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 350–359, Nov 2013.

[15] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 451–462, Nov 2004.

[16] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 621–631, May 2007.

[17] Mark Hennessy and James F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 104–113, New York, NY, USA, 2005. ACM.

[18] John H. Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press, 1975.

[19] Scott E. Hudson, Frank Flannery, C. Scott Ananian, and Dan Wang. *CUP User's Manual.* Graphics Visualization and Usability Center, Georgia Institute of Technology, July 1999.

[20] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, February 2013.

[21] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[22] B. Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, Aug 1990.

[23] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *Testing of Communicating Systems*, volume 3964 of *Lecture Notes in Computer Science*, pages 19–38. Springer Berlin Heidelberg, 2006.

[24] Wes Masri, Rawad Abou-Assi, Marwa El-Ghali, and Nour Al-Fatairi. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, DEFECTS '09, pages 1–5, New York, NY, USA, 2009. ACM.

[25] Wes Masri and Andy Podgurski. An empirical study of the strength of information flows in programs. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 73–80, New York, NY, USA, 2006. ACM.

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

[27] The Open Group. The open group base specifications issue 6 and IEEE std 1003.1: Bc arbitrary-precision arithmetic language, 2004.

[28] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed Random Test Generation. In *In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 75–84, Minneapolis, MN, May 2007.

[29] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[30] J. von Pilgrim, B. Ulke, A. Thies, and F. Steimann. Model/code co-refactoring: An MDE approach. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 682–687, Nov 2013.

[31] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb 2002.

[32] R. Zhao and M.R. Lyu. Character string predicate based automatic software test data generation. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 255–262, Nov 2003.

Vita

Name:	Benjamin Revington
Post-secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada 2007–2013 B.Sc. Honors Specialization in Information Systems and a Major in Statistics
	The University of Western Ontario London, Ontario, Canada 2013–2015 M.Sc. of Computer Science
Honours and Awards:	The Western Scholarship of Excellence, 2007
Related work experience:	Teaching Assistant The University of Western Ontario 2013–2015
	Software Developer Intelliware Software Development Summer 2012

Publications:
none.