Grammar Based Unit Testing for Parsers

Master's Thesis

Jonathan Riehl
University of Chicago
Department of Computer Science

Abstract

White box unit testing is the use of test inputs, derived from source code, to drive test coverage of a metric on that source code. While approaches based on control flow analysis exist for generating unit test inputs, these approaches are not readily applied to code that embeds control flow as static data. One specific example of such codes are the state machines built by parser generators. Control flow in these machine abstractions is based on static data that encodes a set of state transition rules. However, these rules also correspond to the recognition of productions in the grammar input to the parser generator. It is therefore feasible to develop white box tests for a parser based on an analysis of the input grammar.

Using the structure of grammars, a methodology for white box unit testing a parser has been developed and will be presented in this thesis. Furthermore, a tool set that assists in automation of test generation will be described. This tool set is based on the use of a language grammar to create grammatical strings in that language. It will be shown how the set of strings in a language may be navigated in a principled fashion for the purpose of spanning the set of all language productions, and heuristics developed for resolving nonterminal symbols into strings of terminal symbols. A demonstration of how these tools were used to test a simple parser for the C programming language will also be given.

Advisor: David Beazley

1 Introduction

This section will introduce topics that have lead to the development of a unit testing methodology and tool set for parsers. First, traditional methods of white box testing will be illustrated. Second, the operation of parsers and parser generators will be discussed. It will then be shown that the code generated by a common parser generator is not well suited to a unit testing strategy based solely on control flow. This will lead to a discussion of generative grammars, which will play a key role in the proposed solution to the problem of unit testing parsers.

1.1 White Box Testing

Software testing activities may be organized according to a dichotomy based on how test inputs to a program (or subprogram) are developed. In this dichotomy, there are black box tests and white box tests. Black box tests are developed without consideration of a software's source code. In contrast, white box tests are developed using full knowledge of the program's source code.

Black box tests are commonly guided by requirements. For this reason, the aim of many black box tests is to ensure that a subset of the program requirements are met. Since requirements describe program functionality, much black box testing is also known as functional testing. One metric of black box test completeness is requirement coverage. Requirement coverage is a useful metric in traditional software development processes, where the requirements are meant to be decoupled from design and implementation. However, it should be noted that given the definition presented for black box tests, many unit tests, or tests where only parts of programs are tested, are designed to ensure a subprogram provides some base line functionality. These unit tests are not created based on analysis of unit control flow, and do not employ the methodology described for white box testing.

As previously discussed, white box testing is a form of software testing where tests are developed based on structural analysis of the source code. The goal of a set of white box tests is to exercise all control flow paths in a program. Such rigor is employed in order to gain a wider characterization of possible subprogram behavior than is typically exposed by functional testing alone. Where black box testing makes sure code does what it is supposed to do, white box testing helps to ensure that code does not do what it isn't supposed to do. Other methods for gaining characterizations of program failure may also include stress testing or testing boundary conditions specified in

```
function foo (a, b, c):
  if a AND b:
    bar()
  if c:
    baz()
```

Figure 1: Example code showing a function under test.

(or implied by) a design contract. However, only the methodology behind white box testing makes certain guaranties about the amount of code run during testing.

The ideal metric of white box test completeness is full coverage of all possible control flow paths in a program. However, testing on this scale is impractical even for small programs since the number of paths in a program is exponential in the number of control flow branches. Correspondingly, white box tests may be measured using a spectrum of metrics based on how closely the tester is required to approximate full path coverage.

The coursest metric of white box test completeness is based on the number of lines of code run in test. Achieving full line coverage gives the guarantee that every line of code was run under test. One example where the value of line coverage diminishes is given in Figure 1. In the function `foo()`, only one test would be needed to achieve full line coverage, but this does not consider the case where the call to the `bar()` function may have side effects on program state that impact the operation of `baz()` (or even the evaluation of the conditional predicate.) Since no else case is given in the example program, line coverage does not require the tester to look at what happens to the case when a conditional evaluates to false.

A more refined measure of white box test completeness is conditional coverage. Conditional coverage is a measure of the number of true and false pairs run in test for each conditional in the source. Testing at this level of granularity would remedy the fact that testing for full line coverage fails to simulate what happens when a conditional branch is not taken.

A specific deficiency of testing for conditional coverage is caused by the fact that many programming languages use semantics that short circuit conditional truth evaluation. Using these short circuit semantics, the example source code will never evaluate the truth of the b predicate in cases where the a predicate is already evaluated as being false. This can be somewhat remedied by increasing coverage granularity to the level of logical predicates in a conditional. This presents a third level of coverage, one where met-

Level	Target	Example Test Inputs (a,b,c)
1	Full line coverage	(1,1,1)
2	Full conditional coverage	*(1,1,1),* (1,0,0)
3	Full predicate coverage	*(1,1,1), (1,0,0),* (0,1,1) (or also *(1,1,1),* (0,0,0))
4	All predicate permutations	*(1,1,1), (1,0,1), (0,1,1),* (0,0,1), (1,1,0), (1,0,0), (0,1,0), (0,0,0)

Table 1: Example test input at various levels of white box test granularity.

rics measure the number of true and false pairs run for each conditional predicate.

Table 1 illustrates the coverage hierarchy described, with the addition of a fourth level where all logical permutations of predicates are considered. Each level of coverage is followed by example test inputs that would achieve the given coverage goal when input to the `foo()` function in Figure 1. Table 1 also shows how each level of coverage may be looked at as a refinement of the previous level, emphasizing the test inputs that were inherrited from the inputs of the previous coverage level.

Typical white box test methodology involves the selection of a level of coverage. A tester will then analyze features of the target subprogram's control flow at the appropriate granularity. Based on the analysis, the tester will then determine input vectors that will provide the desired coverage. These input vectors are then fed to a subprogram that has been instrumented to generate coverage data, and the coverage results are then verified against the coverage target of each test input.

1.2 Parsers and Parser Generators

A parser is typically a component of an interpreter or compiler and is responsible for performing syntactic analysis on a vector of input tokens. All parsers will ensure that the input token vector conforms to the syntax of a language. Parsers are also commonly used to build a syntax tree or some other structured model of the input vector that is used as input to another component of the interpreter or compiler. While the syntax of a language can be left as implicit in the implementation of the parser, a more common approach is to create a context-free grammar specification for the language [3].

A context-free grammar is useful since it may be used as input to a parser generator. A parser generator will create a parser using some form of context-free grammar input, saving time by creating all necessary logic and control flow for the parser. Some parser generators generate recursive descent parsers, where parser control flow follows the language syntax. However, a sizeable number of parser generators are based on the `yacc` tool.

The `yacc` tool and its successors generate code that simulates a set of state machines. The state machines generated by `yacc` will commonly employ state transition code that is already well understood and tested. The code generated for two input grammars will differ in three ways: the parse table data, the reduction actions, and header and footer code. The parser table data is static data commonly consisting of arrays of integer data. The reduction actions are code segments that are embedded in the input grammar, and are intented to be run when the state machine detects a substring that matches a grammar production. Header and footer code is simply copied from the parser generator input file to the front and end of the generated parser code.

Application of white box unit test methodology to the output of a parser generator is complicated by the fact that control flow is determined in part by the state transition table. As mentioned above, the state transition table is presented in the parser source as an array of integer data. Performing control flow analysis of the parser code will only grant insight into the operation of the state machine excitation logic, not the operation of the parser itself. While the header and footer code may be isolated and tested using white box test methods, these methods do not readily provide a means of targeting the reduction actions without isolating the action case code from the excitation logic. This suggests an approach based on analysis of the reductions in the input grammar would provide some means of targetting parser actions.

1.3 String Generation Using Grammars

Where parsing is the process of determination of a string's membership in a language, the process can also be reversed. Using a context free grammar it is possible to generate strings in the language described by that grammar. Finding strings in the language may be performed by walking a graph that is induced on the structure a context free grammar, G.

An arbitrary context-free grammar, G, may be described as a four tuple, as shown below [8]:

$$G = (N, \Sigma, P, S)$$

$$N = \text{finite set of } \textit{nonterminal symbols} \text{ in } G.$$

$$\Sigma = \text{finite set of } \textit{terminal symbols} \text{ in } G.$$

$$P = \text{finite subset of } N \times (N \cup \Sigma)* \text{ (the set of } \textit{productions.})$$

$$S \in P \quad (= \text{the start symbol.})$$

A context-free grammar may be used to generate strings using derivations. The set of derivations is described by first defining the following binary relation [3]:

$$s_0 \Rightarrow s_1 \iff (s_0 = \alpha_0 A \alpha_1)(s_1 = \alpha_0 \beta \alpha_1)((A, \beta) \in P)$$

A derivation is a sequence of strings, $s_0 \Rightarrow s_1 \Rightarrow \ldots \Rightarrow s_n$, such that $s_i \in (N \cup \Sigma)*$ and $s_i \Rightarrow s_{i+1}$. The common abreviation $s_0 \Rightarrow^* s_n$ is used to imply that there exists a derviation from s_0 to s_n, where $\Rightarrow^*$ is the transitive closure of the $\Rightarrow$ relation.

1.3.1 String Graphs

Using the definition of the single step derivation relation, a directed graph structure, $Gr(G)$, may be defined:

$$Gr(G) = (V(G), E(G))$$

$$V(G) = (N \cup \Sigma)*$$

$$E(G) = \{(v_0, v_1) | v_0 \Rightarrow v_1\}$$

The strings in the language $L(G)$ may be defined as being the set of verticies in $Gr(G)$ that have an out degree of zero and are reachable from S. Generating strings in $L(G)$ can therefore be accomplished by performing walks on $Gr(G)$ from S to strings that are in $\Sigma*$. These walks correspond to derivations, by definition of $E(G)$.

1.3.2 Production Graphs

Another useful graph may be induced on the productions in a grammar:

$$PG(G) = (PV(G), PE(G))$$

$$PV(G) = \{p_0\} \cup \{p_i | p_i \in P\}$$

$$PE(G) = \{(p_0, p_i)|(p_i = (S, \gamma))\} \cup \{(p_i, p_j)|(\exists A \in N)(p_i = (B, \alpha_0 A \alpha_1))(p_j = (A, \beta))\}$$

Later sections will refer to $PG(G)$ as the production graph for grammar G. Walks in $PG(G)$ also correspond to derivations in G, but are constrained such that the next single step derivation replaces only nonterminal symbols inserted by the preceeding derivation step. The vertex p_0 is added to provide a starting point for generating walks of $PG(G)$ that correspond to derivations starting with the grammar's start symbol.

2 Methodology

In Section 1.1 it was shown how coverage metrics drive the kind of code analysis required to generate white box test inputs. By switching the metric, a methodology for white box testing automatically generated parsers becomes simple to develop. In Section 1.2 it was suggested that coverage of productions in the grammar was a useful metric. The following methodology is based on this observation, employing production coverage as one measure of parser white box test completeness.

2.1 A Parser Testing Procedure

By using productions as a coverage metric, the process of white box testing a parser reduces to identifying a target production, creating a test string that will cause the target production to be applied, and verifying that the target action handling code was run in test. Once a means of running the action handling code has been identified, coverage of the action handler code must still meet target coverage levels. Therefore, multiple test inputs may target the same production. In this case, it is left to the tester to determine a means of obtaining full coverage of the action code. When all productions have been fully exercised, testing is complete.

In Section 1.3 it was shown how given an input grammar, G, strings in the language described by G could be automatically generated. Refinements of this technique may be applied to generate test strings in $L(G)$ that are known to exercise specific productions in G. Therefore, the process of generating test inputs that target a specific production may be automated.

The following two step method has been developed to automatically generate unit test strings. The first step involves identification of a string in $(N \cup \Sigma)*$ that is guaranteed to exercise some production in P. The second step involves a walk of $Gr(G)$ starting at the identified string and ending at a string of all terminal symbols. The resulting string is then admitted to the set of white box tests for grammar G. The following two subsections detail each step in this process, and identify improvements to the algorithms sketched above.

2.2 Automation of Production Coverage

The first step in creation of test strings is to generate a string in $(N \cup \Sigma)*$ that excercises some production in P. One very straightforward means of finding these strings would be to employ a bredth first search (BFS) of $Gr(G)$ for a vertex entered by a target production, for each production in P. However, the bredth first search of $Gr(G)$ will commonly cover several productions before a given target production is reached. Redundant computation can be avoided by running a modified bredth first search that searches for a set of targets instead of a single target. The resulting algorithm is presented as pseudo-code in Appendix A.

Besides searching for multiple targets, the bredth first search algorithm given in Appendix A has two notable features. First, the BFS algorithm has been generalized to accept a finite set of constraints, as opposed to a set of target verticies. Second, as a time and space saving technique, the BFS algorithm prunes branches of the search tree once it has been determined that a parent vertex has satisfied an open constraint. Constraints are a required input to this form of BFS because the target strings in $L(G)$ are not known a priori. In practice, the algorithm is extended to output a map from contraints to verticies in $Gr(G)$, mapping from a constraint to the vertex that was first found to satisfy that constraint.

In the most basic case, the set of constraints input to the BFS is "node n is reached by production $p \in P$" for all productions in P. However, in Section 2.1 it was noted that conditionals may be embedded in the code run by a parser when a production is recognized. The inability of the string generator to account for control flow in the target production may be mitigated somewhat by changing the target metric once again.

Control flow in parser action code is typically a function of a state that was determined by the prior sequence of productions already recognized by

1	$S \rightarrow A$
2	$S \rightarrow B$
3	$A \rightarrow a$
4	$B \rightarrow bB$
5	$B \rightarrow c$

Figure 2: An example context-free grammar.

the parser. By changing the coverage metric to the measure of unordered pairs of productions run in test, the automation begins to assist the tester by looking at the side effects one action may have on another. Here the set of constraints becomes "node n such that n is reached by some application of both production p_0 and p_1" for each unorderred pair of productions, $(p_0, p_1), p_0, p_1 \in P$. Since $|P|(|P| - 1)/2$ additional strings are generated, this exapanded set of constraints may also be seen as providing a stress aspect to the generated test strings.

This refinement on the input constraints is not without problems. First, it ignores the fact that inter-production interactions in the parser state are almost certainly assymmetric (implying *ordered* pairs of productions are a better coverage metric). Second, it adds the possibility of the BFS diverging, since some input grammars may define mutually exclusive productions causing some of the input constraints to never be met. Both of these issues are a reflection of the recursive nature of grammars.

Ordered pairs of productions may be described as constraints to the BFS algorithm by changing the pair constraint to read "node n such that n is reached first by application of production p_0, then production p_1". However, this ignores that the grammar may not have enough recursion to support the given order of production application. Figure 2 shows an example grammar where application of production 4 will never follow application of production 5. Typical parser grammars will provide enough recursion that this is overcome, but the case where a language has a header section or some other partition of the grammar is still common enough to preclude the use of ordered pairs (such as a top level declarator section, followed by a top level definition section).

Figure 2 also illustrates a grammar where two productions are mutually exclusive. In this case, production 1 is mutually exclusive with production 2 (which in turn forces production 3 to be mutually exclusive with productions

4 and 5.) It is straightforward to see that using S as the start symbol forces one to either apply production 1 or production 2. Since the grammar is not recursive with respect to the S symbol, S will never reappear in the intermediate strings generated either production 1 or 2. Therefore, both productions will may ever be applied to the same string in $L(G)$. However, typical parser grammars will be recursive enough that this will not happen, containing a production or set of productions similar to "$S \rightarrow SS$" (such as a compound statement or a compound definition production).

2.3 Resolution of Nonterminal Symbols to Strings of Terminals

Once a set of strings in $(N \cup \Sigma)*$ are found, it remains to resolve these strings into strings of all terminal symbols, $\Sigma*$. One solution would be to iterate over each string, performing some search in $Gr(G)$ starting at the string, and ending at the first vertex in $Gr(G)$ that has no out degree (by definition of $E(G)$, these are verticies that correspond to strings in $\Sigma*$.) This approach is inefficient because each walk in $Gr(G)$ could follow the same set of productions for common nonterminals in the strings.

It is sufficient to build a map from the set of nonterminals, N, to a subset of strings in $\Sigma*$. Such a map may be built by iterating over the set of nonterminal symbols, N. For each nonterminal symbol, A, some search over $Gr(G)$ is performed. The search would start at the vertex corresponding to the string consisting only of the nonterminal, A, and ending at some vertex with an out degree of zero. Further time may be saved by memoization of prior search results, replacing nonterminals in a working string with the substring corresponding to the result of the search for the known nonterminal symbol.

During development of the initial prototype, a BFS algorithm was initially used for the search algorithm. However, this consumed all system memory for the C++ grammar under consideration. The grammar presented in Figure 2 presents an example of why a depth first search (DFS) will also fail for most input grammars. Assuming The DFS algorithm is employed, recursion in the input grammar may cause DFS to diverge. Therefore some sort of heuristic search is required.

The heuristic search is similar in form to the BFS algorithm, but for each vertex a cost is calculated. When a vertex is removed from the frontier, only verticies of minimal cost are chosen. By using the length of the symbol string associated with each vertex as a measure of cost, the heuristic search is capable of finding a map from nonterminal symbols to strings of all terminal symbols in a reasonable amount of time and within an appropriate space bound.

3 Design and Implementation

The test string generation algorithm outlined in Section 2 is intended to assist language implementors perform white box tests on parsers they develop. It was therefore deemed beneficial to integrate the test string generator into a programming language development framework. This section will provide a high level description of the Basil language framework, show how the test string generation application fits into the software framework, and will demonstrate how the application was used to test another component of the framework.

3.1 The Basil Language Framework

The Basil language framework is a Python [17] based set of libraries and applications that are intended to assist software developers in two ways. First, the framework is meant to provide a common infrastructure for programming language implementation. Second, the framework should provide reusable components that allow the development of cross language software engineering and analysis tools. These goals are met by providing components that are customized to support development of specific portions of the compiler pipeline.

The framework is organized into four sets of components. Components that are built around a target data representation are called integrations. There are three sets of integrations: parser generator integrations, language integrations, and model integrations. The fourth set of components are applications. The set of Basil applications is a catch all for tools present in the framework that employ the functionality of one or more integrations, but are not deemed as being a part of a specific integration.

3.1.1 Parser Generator Integrations

A parser generator integration is a component that translates from the input language of some parser generator to a grammar model. A parser generator integration may also provide some means for translation of a grammar model back to a parser generator input file, but this is not a requirement. A parser generator integration will typically consist of a parser for the parser generator input, a translation back end that walks a syntax tree of the input file and generates a grammar model, and an optional model walker that walks a grammar model instance and generates a input source for the target parser generator. Examples of parser generators that are targetted for integration are yacc and Python's parser generator, pgen. Parser generator integrations may also reimplement or couple more tightly to parser generators to assist with the development of interactive grammar development tools.

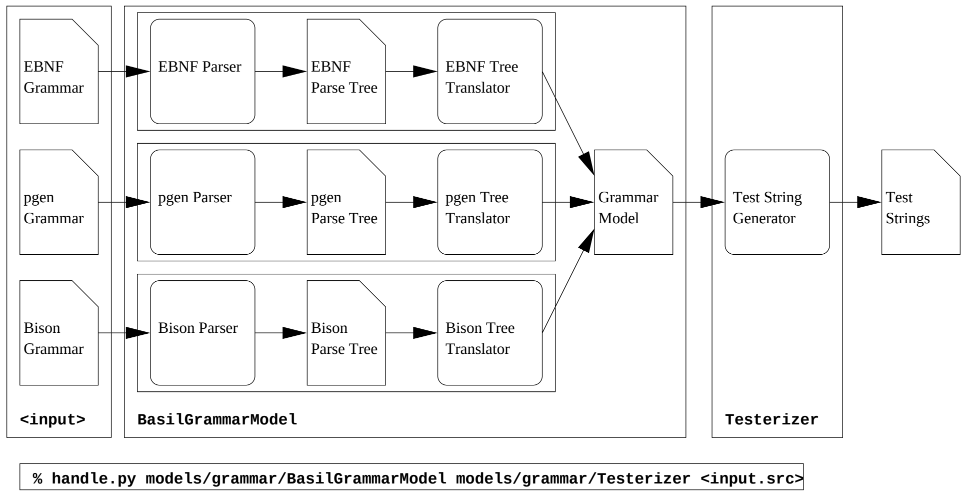
3.1.2 Language Integrations

A language integration is essentially a parser that translates a source string to a syntax tree for some predefined language. Language integrations target a syntax tree model that employes native Python data structures for speed purposes. This may be contrasted to other model integrations which use objects to represent model instances. A standard model integration incurs time and space overhead issues because of the large number of model elements present in a parse tree. Language integrations may also include facilities to generate source code, but in many cases it is simpler to output code as text to the console or a source file, versus writing a tree walker to output code.

3.1.3 Model Integrations

A model integration either defines or is generated from a data structure description, referred to as a meta-model. The meta-model is exposed as a set of classes and a factory class. The factory class is responsible for model serialization and deserialization, as well as exposing model element construction methods. The goal of a model is to serve as an inmemory representation of some intermediate language. Some examples of target model domains would be object models (MOF, UML), linkage interface models (SWIG), control flow graphs, data flow graphs, and the Basil grammar model.

The use of class based model representations allows meta-model development to employ third party object oriented modeling tools [15] [16]. Class based models also allow models to contain cyclic references, structuring data as directed graphs. This may be contrasted with hierarchical data organizations such as inmemory representations of XML, which force a model developer to employ indirect references to other model elements. However, as was mentioned in Section 3.1.2, the overhead of object construction and navigation can be expensive, and Basil also provides a tuple based facility for creating, storing and manipulating tree data structures.



```
% handle.py models/grammar/BasilGrammarModel models/grammar/Testerizer <input.src>
```

Figure 3: Dataflow of the Test String Generation Application

3.1.4 Applications

Applications are framework components that will typically provide translation facilities from one model domain to another. One example application would be a component that walks a grammar model instance and outputs a model integration, thus creating a parse tree model specific to the language specified in the grammar model. By employing an arbitrary parser generator integration to build the grammar model instance, the application would allow model integrations to be automatically built for all inputs for all supported parser generators. In cases where an application is useful in the development of the Basil framework, it will often be exposed as a program that may be run from the OS shell.

3.2 The Basil Test String Generation Application

Figure 3 illustrates the components and data products that comprise the bulk of the test string generation application. Also shown is an example of how to use the command line interface. The command line interface uses the `handle.py` program. The `handle.py` program is a Basil application that provides the glue for building compilers in Basil. Application back ends are run from `handle.py` by specification of an intermediate language (as served by a model integration), the application back end (also termed a *handler* of the input model), and an input source file.

When a model factory is asked to parse an input file and construct a new model instance, the factory will dispatch to a parser or front end based on the file extension given in the input file name. For the test string generator, the Basil grammar model is employed. When `handle.py` is run, the grammar model dispatches to a parser/handler pair. The selected parser will build a parse tree that is then input to the handler. The handler will walk the parse tree, and create a new grammar model instance. Finally the new model instance is fed to the top level handler.

The top level handler for the test string generator implements the algorithms described in Section 2. The generator begins operation by extracting the set of productions in the input grammar and a start symbol (the first nonterminal symbol listed in the model is used if none is present.) Using the start symbol and the set of productions, only parts of the infinite di-graph, $Gr(G)$, are instantiated as the modified BFS is run. A map from all nonterminal symbols to strings of terminal symbols is generated, then applied to any nonterminals found in the strings generated in the first part. The resulting test strings are then output to the terminal. Optionally, the set of productions targeted by each string and the map from nonterminal symbols to terminal strings may also be output.

3.3 Example Usage

Using the test string generator, it becomes possible to develop white box test strings for languages supported by the framework. For example, the C language integration uses a parser generated by Bison. Since a Bison parser generator integration is available, the test string generator may be used to generate test input for the C language integration. These strings are generated using the following command:

```
% handle.py models/grammar/BasilGrammarModel \
            models/grammar/Testerizer \
            lang/c/cparser.y -verbose > c-test-strings.txt
```

The resulting strings generated are shown as follows:

```
0. CONST ';' CONST ';'
1. CONST ';' CONST ';' CONST ';'
2. CONST ';' IDENTIFIER '{' '}'
3. CONST ';' CONST ';'
4. IDENTIFIER CONST ';' '{' '}'
...
```

The index numbers allow the tester to cross reference the target production list with the generated string. For example:

```
...
0. ('external_declaration', 'external_declaration')
[(0, ('translation_unit', ('translation_unit',
                           'external_declaration'))),
 (0, ('translation_unit', ('external_declaration',))),
 (0, ('external_declaration', ('function_definition',)))]
...
```

The first tuple is the string as originally phrased in mixed terminal and nonterminal symbols. The list of tuples following the string identifies the set of productions that are being targetted by that string. The tester may verify these productions were exercised by the given string using a coverage analysis tool such as `gcov`.

In order to use these test cases the tester must map from nonterminal symbol names to actual lexical entities. The resulting strings must then be input to an instrumented parser that allows verification of coverage. For example, the Basil C parser can be compiled as a command line program that accepts C source from the console. The following describes how the test methodology was applied to the Basil C parser.

Using the "`-fprofile-arcs`" and "`-ftest-coverage`" arguments to `GCC`, the comand line program was instrumented. The strings were then copied to files that were piped to the instrumented parser. During parsing, the parser instrumentation generates output files, which are then analyzed using `gcov`.

```
% make "CFLAGS=-g -fprofile-arcs -ftest-coverage" cparser
[ Builds an instrumented C parser ]
% cparser < tests/test6.0.in
[ Runs the instrumented C parser,
  generating instrumentation data files ]
% gcov cparser.tab.c
[ Creates file cparser.y.gcov ]
```

Since Bison generates source with the C `#line` pragma, `gcov` is able to trace coverage data directly back to the original Bison input file (thus generating a "cparser.y.gcov" file as opposed to "cparser.tab.c.gcov", which could be more difficult to read.) The lines of "cparser.y.gcov" that correspond

```
...
              translation_unit
                      : external_declaration
                      {
         1              $$ = $1;
         1              CParserSetRoot($1);
                      }
         1            | translation_unit external_declaration
              {
         2      $$ = CParserNewNode(TRANSLATION_UNIT, 2);
         2      CParserSetChild($$, 0, $1);
         2      CParserSetChild($$, 1, $2);
         2      CParserSetRoot($$);
              }
         2            ;
...
```

Figure 4: Partial output of `gcov` for an instrumented run of the Basil C Parser

to those targetted by the selected test string are shown in Figure 4. The numbers in the left margin count the number of times the given line was run by the instrumented parser. Not shown in Figure 4 are lines that were not run in test. Lines not executed in test would have a series of hash marks in the left margin to mark their untested status. In a full test situation, the tester would continue by identifying productions not tested, and apply the same methods shown above until all code was shown to have run in test.

4 Related Work

Various approaches to white box testing, also termed as structural testing are described by Jorgensen [7]. Specifically, Jorgensen attributes development of coverage metrics to Edward F. Miller [12], who developed a lattice of coverage metrics similar to those described in Section 1.1. Thomas McCabe developed metrics and tools to support both flow control analysis and coverage instrumentation of software [18]. A more recent survey of attempts to automate structural testing can be found in Edvardsson [5]. In his survey, Edvardsson mentions sytax driven test case generation, but only to comment that is it outside the scope of the paper.

String generation from grammars begins with Chomsky's fomalization of grammar, where he describes his formalism as being "a device of some sort for producing the sentences of the language under analysis" [4]. Grammar based string generation has been widely used in the creation of procedural graphics. Lindenmayer is well known for using generative grammars as a means of describing and procedurally generating plant structure [13]. Generative grammars have also been applied to generation of music [10]. Peter Maurer's Data Generation Language (DGL) generates strings using an input grammar, which have been used in circuit testing [11].

Most closely related to the work done here is the work of Paul Purdom [14]. In [14], Purdom decomposes the problem of testing parsers in a fashion similar to Section 2.1, where an algorithm is given to find a mapping from nonterminal symbols to strings of terminal symbols, and a second algorithm is used to find a set of derivations that cover all productions in a grammar. Purdom's high level methods are geared toward verification of parser correctness, rather than acting as a substitute for structural testing of a parser. Another important difference is Purdom's algorithm for mapping nonterminal symbols to strings of terminals is much more efficient than the one given in Section 2.3, and should be adopted in a future release of Basil. In [9], Lämmel extends Purdom's techniques with a new metric based on coverage of all edges in the production graph, $PR(G)$, as defined in Section 1.3.2.

Grammars appear again in the field of model checking, which is related to functional testing. In model checking, software is modeled as automata while requirements are specified as logical assertions about automata behavior. Both the logical assertions about model correctness and the automaton model can be reformulated as grammars. A model checker may then test for intersection of the complement of the assertion language and the automata language, and ensure the intersection of the two languages is empty. One such model checker is SPIN, developed at Bell Labs by Gerard Holzmann [6]. Research into using source code to create model automata (thus moving model checking closer to structural testing) is also being undertaken. This requires expansion of the kind of admissible models, from finite automata to a subclass of infinite automata [1], and logics capable of making assertions about the expanded automata models [2].

5 Future Directions

The demonstration of toolset usage provided in Section 3.3 illustrates points where automation may be further enhanced. Ideally, the loop of string generation and coverage verification could be automated, but in a highly platform dependent fashion (for example, scripting of the test run and `gcov` output validation would be specific to parsers generated by Bison on a platform with `gcov` and GCC.) It would also be desirable to automate the translation from terminal symbol names to actual lexical strings, even if not all terminal symbols would not have a clear one-to-one mapping (such as identifiers.) Another step would be to allow some subset of semantics to be associated with productions. Therefore if the test string generator emitted a C string with an identifier, the tester could have the tool insert a declaration of the identifier before its use.

The methodology as presented is focused on generation of strings in some language, $L(G)$, for the purpose of driving coverage of source code under some metric. However, if the stated purpose of these exercises is to ensure a parser/compiler *does not do what it is not supposed to do*, it is not a complete means of addressing common failure cases inherrent in language implementation. Therefore, a more robust test methodology should look at not only how a parser behaves given strings in the language but how well it recovers given strings that are agramatical. Agrammatical strings would be encompassed by use of a metric that measured coverage of a production as well as coverage of an error case for that production. The generator would accomplish this by misapplication of the production, such as omitting or inserting a symbol. One problem with this method is the parsing algorithm used could complicate direct correlation between an intended error in a production and the error actually exercising error handling code for a production.

Furthermore, programming language test methodology should distinguish between tests of the language that are grammatical, but are either semantically invalid, or semantically valid. Language formalisms that take the form of context sensitive grammars may assist with development of not only a superset of the methodology presented, but also extension of the test automation. Under the adapted methodology, the target coverage metric would measure counts of both violation and correct usage of all semantic productions under test. Generation of these inputs would make the automation more like an automated theorem generator, and such tools would be able to employ common techniques used in theorem discovery.

6 Conclusion

White box testing of parsers is complicated by the nature of common parser implementation techniques. Parsers that use state machines will have control flow hidden from standard code analysis techniques. Instead of language level control flow constructs, the generated parser encodes state transitions in lookup tables, and state actions in flat case statements. However, the structure of the state machine follows the structure of the language it is intended to recognize, and this fact may be exploited through analysis of the language's grammar. By modification of the coverage metric considered by the white box test methodology, control flow is abstracted to focus on recognition of productions in the grammar. At this level of abstraction, tests focus on excercising the language implementor's code, and not the generic state machine code developed for use by a parser generator.

By using a language's grammar as a generative grammar, automation of the modified methodology becomes feasible. Search techniques and a useful heuristic have been illustrated. The algorithms have been employed in an application that assists in the development white box test cases for a language parser. It is hoped this application will be of particular value in the context of a programming language prototyping framework, allowing domain specific languages to not only be rapidly developed, but rapidly tested as well.

References

[1] Alur, Rajeev, and P. Madhusudan. *Visibly Pushdown Languages.* STOC 04. Chicago, 2004. Association for Computing Machinery, *to appear.*

[2] Alur, Rajeev, Kousha Etessami, and P. Madhusudan. *A Temporal Logic of Nested Calls and Returns.* TACAS 2004. Barcelona, Spain, 2004. Heidelberg: Springer-Verlag, 2004.

[3] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools* Reading, Mass.: Addison-Wesley Publishing Company, 1986.

[4] Chomsky, Noam. *Syntactic Structures* The Hague: Mouton, 1976.

[5] Edvardson, Jon. "A Survey on Automatic Test Data Generation" Proceedings of the Second Conference on Computer Science and Engineering in Linkping, CCSSE'99, ProNova, Norrkping, 1999.

[6] Holzmann, Gerard. “The Model Checker SPIN.” *IEEE Transactions on Software Engineering* 23.5 (1997): 279-295.

[7] Jorgensen, Paul C. *Software Testing: A Craftsman's Approach.* Boca Raton: CRC Press, 2002.

[8] Kozen, Dexter C. *Automata and Computability* New York: Springer-Verlag, 1997.

[9] Lämmel, Ralf. “Grammar Testing.” *Proceedings of Fundamental Approaches to Software Engineering.* FASE 2001. LNCS v.2029. New York: Springer-Verlag, 2001.

[10] Lerdahl, Fred, and Ray Jackendoff. *A Generative Theory of Tonal Music* Cambridge, Mass.: MIT Press, 1983.

[11] Maurer, Peter M. “The Design and Implementation of a Grammar-based Data Generator.” *Software - Practice and Experience* 22.3 (1992): 223-244.

[12] Miller, Edward F. Jr. *Tutorial: Program Testing Techniques.* COMPSAC 77. Chicago, 1977. IEEE Computer Society, 1977.

[13] Prusinkiewicz, P., and A. Lindenmeyer. *The Algorithmic Beauty of Plants* New York: Springer-Verlag, 1990.

[14] Purdom, P. “A Sentance Generator for Testing Parsers.” *BIT* 22.3 (1972): 366-375.

[15] Quatrani, Terry. *Visual Modeling With Rational Rose and UML.* Reading, Mass.: Addison-Wesley Publishing Company, 1997.

[16] Robbins, J. E., D. Hilbert, and D. Redmiles. “Extending Design Environments to Software Architecture Design.” *Automated Software Engineering: An International Journal* 5.3 (1998): 261-290.

[17] van Rossum, Guido. *Python Reference Manual* Ed. Fred L. Drake, Jr. Python Labs, 2003.

[18] Watson, Arthur H., and Thomas McCabe. *Structured Testing : a Testing Methology Using the Cyclomatic Complexity Metric.* Ed. Dellores Wallace. Gaithersburg, Maryland: NIST, Special Publication 500-235, 1996.

A A Modified Bredth First Search

```
Inputs:
    T     = Finite set of target constraints.
    E     = Map from nodes to edges.
    start = Start symbol of grammar.
Outputs:
    S = Set of found targets.
Begin:
    Frontier = {start}
    Visited = {}
    S = {}
    while (T is not empty) and (Frontier is not empty):
        n = Pick node from Frontier
        Remove n from Frontier
        Add n to Visited
        for edge in E(n):
            dest = Node targetted by edge.
            if (dest not in Visited) and
               (dest not in Frontier):
                    if dest satisfies some constraint, c, in T:
                            Remove c from T
                            Add dest to S
                            Add dest to Visited
                    else:
                            Add dest to Frontier
    return S
End.
```

Figure 5: Bredth first search modified to find multiple targets.