A Systematic Framework for Grammar Testing [*]

Lixiao Zheng[1,2], Haiming Chen[1]
[1]State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
[2]Graduate University, Chinese Academy of Sciences
{zhenglx,chm}@ios.ac.cn

Abstract

Grammars, especially context-free grammars, are widely used within and even outside the field of computer science. In this paper, we present a systematic framework for grammar testing, in which some commonly used techniques for testing programs such as module testing and integration testing are adapted and applied to the testing of grammars. We propose a nonterminal-based approach for grammar modularization, combined with an iterative process for grammar testing in which a grammar is tested with respect to both a generator and a recognizer. Experiments on grammars for some non-trivial programming languages such as C and Java demonstrate the feasibility and efficiency of the testing framework and the proposed approaches.

1. Introduction

Grammars, especially context-free grammars, are useful in computer science and software development. The importance of grammars in language design and implementation is well known. Besides the above, grammars can be found in many other applications. For example, [7, 10] presented a recent account of many applications of grammars. Such systems are called *grammar-based systems* [7] or *grammarware* [10]. Despite the pervasive role of grammars in software systems, Klint, Lämmel and Verhoef [10] noted that "In reality, grammarware is treated, to a large extent, in an ad-hoc manner with regard to design, implementation, transformation, recovery, testing, etc." They called this phenomenon as grammarware hacking. Therefore sound and systematic methods and techniques are needed for grammarware to move from hacking to engineering.

This paper tackles the issue of grammar testing from an engineering perspective. We present a systematic testing framework which adapts techniques for testing programs in traditional software engineering to the testing of grammars. In particular, we adapt concepts such as static/dynamic testing and module/integration testing, which are commonly used in software testing, and apply them to grammars. We propose an approach to partitioning a grammar into subgrammars, called modules, based on which module testing and integration testing are performed. We propose a novel method to test grammars in which grammars are checked, and if not correct, iteratively revised to approach correct ones, by sentence generation, sentence recognition and coverage analysis techniques. This method tests a grammar from both aspects of its functionalities, i.e., a generator and a recognizer, offering more adequacy and efficiency. We implemented the framework and evaluated it by the application to grammars for two mainstream programming languages, C and Java. The experimental results are promising.

The rest of the paper is organized as follows. Section 2 provides some background propaedeutics. Section 3 gives an overview of the testing framework. In Section 4 and Section 5 we detail the approaches to grammar modularization and grammar testing respectively. Experimental results are then illustrated in Section 6. Section 7 reviews the related work and Section 8 concludes the paper.

2. Background

In this section we describe some terminologies associated with context-free grammars and then give a brief explanation of grammar testing.

2.1. Terminology

We choose context-free grammar as the grammar formalism in this paper. Formally, a *context-free grammar* (CFG) is a 4-tuple $G = (N, T, P, S)$ with N and T finite disjoint sets of *nonterminals* and *terminals* respectively, P a finite set of *productions* of the form $A \to \alpha$ where $A \in N, \alpha \in (N \cup T)^*$, and $S \in N$ the *start symbol*. For a production $A \to \alpha$, we occasionally say A is *defined* by α. A *derivation step* is an element of the form $\gamma A \beta \Rightarrow \gamma \alpha \beta$

[*]This work is supported by the National Natural Science Foundation of China (NSFC) under grant numbers 60573013, 60721061.

with $\gamma, \beta \in (N \cup T)^*$ and $A \to \alpha \in P$. A *derivation*, denoted as $\overset{*}{\Rightarrow}$, is a sequence of derivation steps. The language defined by G, denoted as $\mathcal{L}(G)$, is the set of strings $\mathcal{L}(G) = \{w \in T^* | S \overset{*}{\Rightarrow} w\}$. A string w is called a *sentence* of G, if $w \in \mathcal{L}(G)$. A nonterminal A is *reachable* if there exist $\gamma, \beta \in (N \cup T)^*$ such that $S \overset{*}{\Rightarrow} \gamma A \beta$. A is *productive* if there exists $w \in T^*$ such that $A \overset{*}{\Rightarrow} w$. A grammar is *reduced* if it has neither unreachable nor unproductive nonterminals. A *parse tree* is an ordered tree whose root is the start symbol, leaf nodes are terminals and interior nodes are nonterminals. The children of any non-leaf node A correspond precisely to those symbols on the right-hand side of a production for A. A grammar is *ambiguous* if there exist more than one parse tree for one sentence.

Without loss of generality, we assume in the rest of the paper that the start symbol S does not appear in the right-hand side of any production.

2.2. Grammar Testing

A grammar defines a language and provides a device for generating sentences. Ensuring that a grammar specifies an intended language is indeed a validation problem. In this case, the object under test is a grammar G and the requirement or specification that G is tested against is an intended language L. Consequently, the purpose of grammar testing is to validate whether the defined language meets the user's requirement and to find potential errors which cause it to specify some language other than the one intended.

In general, there are two classes of faults with a grammar G with respect to an intended language L [5].

- incorrect. A grammar G is said to be *incorrect* with respect to an (intended) language L if $\mathcal{L}(G) \nsubseteq L$.
- incomplete. A grammar G is said to be *incomplete* with respect to an (intended) language L if $L \nsubseteq \mathcal{L}(G)$.

It should be noted that incorrectness and incompleteness are usually interwinded [5]. An incorrect phase often causes some incompleteness and correcting the phase also contributes to the completion of the grammar. The process of grammar testing is to repeatedly find incorrectness and incompleteness errors and eliminate them until the final grammar resembles the structure of the intended language.

3. The Testing Framework

There are two classes of approaches to software testing [8]. Reviews, walkthroughs or inspections are considered as static testing, whereas actually executing programmed codes with test cases is referred to as dynamic testing. A common practice of dynamic testing is usually performed on different levels: module testing, a process of testing individual subprograms; integration testing, exposing defects in the interfaces and interactions between integrated components (modules); and other higher-level testing. We adapt some of these techniques to grammars. We interpret static testing as static checks on grammar definitions. As for dynamic testing, we consider a grammar as consisting of a set of sub-grammars, just like a program as consisting of subprograms, and introduce the concept of module/integration testing into grammar testing. Figure 1 gives an overview of the testing framework.

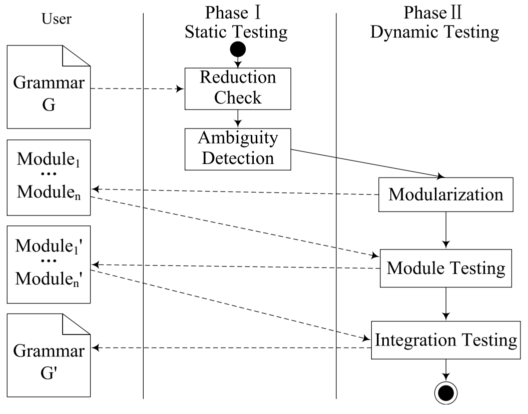


Figure 1. The testing framework.

The testing begins with static inspections on the grammar, including reduction check and ambiguity detection. The former targets at checking whether there exist unreachable or unproductive nonterminals. This helps to find some definition errors that may be caused by the definer's incaution. For example, if a nonterminal is defined but not used, the fact is most likely that this nonterminal is indispensable but is forgotten to be used in some productions. Ambiguity is not necessarily an error but it might be in some cases expected to be avoided. Thus ambiguity detection is also useful and should be included in this phase.

In phase II the grammar is first modularized, then module testing and integration testing are performed. The benefits of modularization for testing, especially for testing large-scale grammars, are significant. It makes testing more simple, precise and efficient. Note that here the step of modularization is necessary because unlike programs whose modules are already explicit before testing, a grammar is usually defined in a big-bang style without a clear notation of modularity. Indeed, there exist some grammar notations supporting modular syntax definition, such as SDF [3]. Unfortunately, it is not the case for most commonly used ones, like BNF and EBNF.

Computing the set of unreachable and unproductive nonterminals is not difficult [4]. But the ambiguity problem of general context-free grammars is known undecidable. Fortunately, there have been some conservative algorithms [13] which can tackle this problem approximatively and can be adopted in our framework. The remaining problems are modularization and testing method which we will discuss in detail in the next two sections.

4. Modularization

From the software engineering perspective, a module should be high cohesion and loose coupling. For a grammar G, a module is in fact a sub-grammar of G. Therefore grammar modularizing is essentially a process of partitioning a grammar into sub-grammars, with each sub-grammar having frequent interactions internally but fewer interactions with each others. The partition may be either production-based or nonterminal-based, and the interactions may be represented by well-defined relations between productions or nonterminals, accordingly. The former divides the production set of a grammar into disjoint subsets. In this approach, two productions with the same left-hand side may fall into different blocks. Since we want to make sure that when a nonterminal is tested, the information attached to it is as complete as possible, we hence prefer the nonterminal-based partition, which is defined as follows.

Definition 1 *Let $G = (N, T, P, S)$ be a CFG. $\{G_i = (N_i, T_i, P_i, *)\}_{i=0}^{n}$ is called a partition of G, where $\cup_{i=0}^{n} N_i = N$, $N_i \cap N_j = \emptyset, i \neq j$; $P_i = \{p | p \in P, lhs(p) \in N_i\}$; $T_i = \{t | t \in T, t$ has occurrence in $P_i\}$.*

To construct sub-grammars from the partition, we introduce a notion *virtual terminal* which maps each nonterminal to a terminal prefixed with vt. A virtual terminal is essentially a nonterminal but acts as if it were a terminal [15].

Definition 2 *Let $G = (N, T, P, S)$ be a CFG. $\{G'_i = (N'_i, T'_i, P'_i, S'_i)\}_{i=0}^{n}$ is called the sub-grammars of G with respect to partition $\{G_i = (N_i, T_i, P_i, *)\}_{i=0}^{n}$, where $N'_i = N_i$; $T'_i = T_i \cup \{vt_A | A \notin N_i$, A has occurrence in $P_i\}$; $P'_i = \{p' | p \in P_i, p'$ is obtained from p by replacing each occurrence of A by vt_A for all $A \notin N_i\}$; $S'_i \in N_i$. In particular, it is required that there exists a G'_i with $S'_i = S$.*

As can be seen from the above definitions, the crucial steps of grammar modularization are first the partition of nonterminals, then the selection of start symbol for each sub-grammar. And last, to serve for the subsequent integration testing, a derivation of relationship between sub-grammars is also needed. Partition of nonterminals is the key point in the whole process. We accomplish the three steps by utilizing the following dependency relation $\rhd$ defined on nonterminals.

Definition 3 *Let $G = (N, T, P, S)$ be a CFG. For any $A, B \in N$, A depends on B, denoted as $A \rhd B$, iff there exists $p \in P$ such that $A = lhs(p)$ and B has occurrence in $rhs(p)$.*

Step 1: Partition of Nonterminals

From the dependency relation, we can construct a digraph whose nodes are nonterminals and where there is an edge from A to B precisely when $A \rhd B$. A first idea that comes to mind is hence to partition the dependency graph into strongly connected components (SCCs). Since each SCC internally forms a complete graph, we may assume a high degree of interdependence between the nonterminals in a given SCC and consider this as an indicator of "high cohesion". In fact, this digraph and its SCCs have been used in [11] to measure the structure complexity of grammars. Unfortunately, as revealed by the results in [11], many of the SCCs derived from the digraphs are in fact of size 1. Apparently, a large number of singleton SCCs might result in a high degree of dependence between each others, which is opposite to our "loose coupling" target. To solve this problem, we take a further step to group singleton SCCs.

Our partition algorithm utilizes an important property of DAGs. A DAG is a Directed Acyclic Graph, namely, a directed graph without directed cycles.

Proposition 1 *In a DAG, there always exists a node that has no incoming edges, i.e., with in-degree 0.*

The proof of this proposition is very simple and will not be presented here due to space limitation.

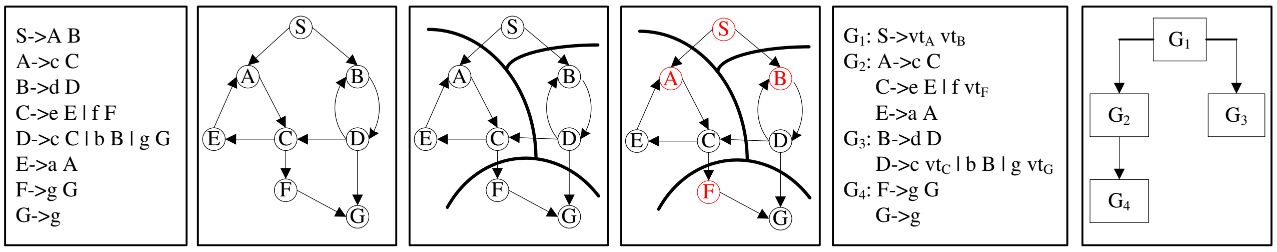
Algorithm 1: Partition of Nonterminals

```
input  : a grammar $G = (N, T, P, S)$
output: a partition of nonterminals of grammar $G$
1  begin
2      $\mathcal{PN} \longleftarrow \emptyset$
3      build the dependency graph $\mathcal{G}$ from $G$
4      compute the $SCCs$ of $\mathcal{G}$
5      for each non-singleton $scc \in SCCs$ do
6          $\mathcal{PN} \longleftarrow \mathcal{PN} \cup \{scc\}$
7          remove $scc$ and its associated edges from $\mathcal{G}$
8      repeat
9          find a node $v$ with $indegree(v) = 0$
10         $V \longleftarrow \{v\} \cup \{u | u$ is reachable from $v\}$
11         $\mathcal{PN} \longleftarrow \mathcal{PN} \cup \{V\}$
12         remove $V$ and its associated edges from $\mathcal{G}$
13     until $\mathcal{G}$ is empty
14 end
```

Algorithm 1 describes the basic structure of the partition procedure. Line 5 through 7 pick out the non-singleton SCCs, mark them as parts of the partition, then remove them, along with all the associated edges, from the dependency graph. The removal of non-singleton SCCs leaves the graph an acyclic one[1], where, by proposition 1, we can find at least one node, namely the *source node*, that has no incoming edges. Collect all the nodes that can be reached from the source node, mark this collection as a part of the partition, then similarly, remove it from the dependency graph. This process, line 9 through 12, is repeated until the dependency graph is empty.

The subsets of nonterminals in the final partition fall into two categories: SCCs, in which any two nonterminals have a direct or indirect dependency relation, and non-SCCs, in

[1]Note that here we ignore all the self-loop edges, i.e., edges that connect a node with itself. This does not impact the final partition result.



(a) The grammar (b) Dependency graph (c) Partition (d) Selection (e) Modules (f) Call graph

Figure 2. An example of grammar modularization.

which any nonterminal is depended, directly or indirectly, by the source nonterminal. The partition condition is weakened from the former to the latter for purpose of a good degree of modularity.

Step 2: Selection of Start Symbols

As we have mentioned, there are two categories in the final partition result, SCCs and non-SCCs. For each non-SCC, it is reasonable to select the source node as its start symbol. However, things become complex when it comes to SCCs since all the nodes in a given SCC are reachable from each other and more opportunities exist for the selection of start symbol. In our approach, we restrict the candidates into an interesting set of *entry points*.

Definition 4 *Let $G_1 = (N_1, T_1, P_1, *)$ be a part of partition of grammar G, the entry points of G_1 is the set of nonterminals $\{A | A \in N_1, \exists B \notin N_1$ such that $B \rhd A\}$.*

We then choose the start symbol for each SCC sub-grammar from its entry points. Again, the choice makes use of the dependency graph, as described in Algorithm 2 from line 5 to line 11. First we build the dependency graph on the entry points, then collect the nodes that have no incoming edges and pick the one that is defined first in the grammar as the start symbol (line 9). If no such nodes are found, which means there exist direct cycles in the graph, then pick one from the entry points, using the same *pick* strategy (line 11). The strategy that pick the one defined first comes from the consideration that a grammar is usually defined from large concepts to small concepts and a large concept is more appropriate to be the sub-grammar's start symbol. For example, when defining a programming language, we usually first define the concept "program", then define its components such as "declarations", "statements" and then go on with even smaller concepts.

By assumption, the start symbol S does not appear in the right-hand side of any production, indicating that S has no incoming edges in the dependency graph. Thus S will be chosen as a source node for a sub-grammar in the partition step and will then be selected as the start symbol for that sub-grammar. This exactly satisfies the requirement in Definition 2 that there should exist a G'_i with $S'_i = S$.

Step 3: Derivation of Relation between Modules

Algorithm 2: Selection of Start Symbols

```
input : a partition of nonterminals $\{N_i\}_{i=0}^n$ of $G$
output: a selection of start symbols $\{S_i\}_{i=0}^n$
1  for $i \leftarrow 0$ to $n$ do
2      if $N_i$ comes from non-SCCs then
3          $S_i \longleftarrow$ source node of $N_i$
4      else
5          compute the entry points $EP_i$ from $N_i$
6          build the dependency graph $\mathcal{G}_{EP_i}$ of $EP_i$
7          $V_i \longleftarrow \{v | indegree(v) = 0\}$
8          if $V_i \neq \emptyset$ then
9              $S_i \longleftarrow v, v \in V_i, v$ is defined first in $G$
10         else
11             $S_i \longleftarrow v, v \in EP_i, v$ is defined first in $G$
```

In our testing framework, we parallel a grammar to a program and the partitioned sub-grammars to the procedures or subprograms that construct the program. In particular, since a sub-grammar defines a language whose sentences are strings derived from the sub-grammar's start symbol and consisting of the sub-grammars's terminals and virtual terminals, we may regard the sub-grammar as a procedure whose input are the virtual terminals, output is its start symbol and functionality is to specify the derivation of valid strings. Based on this consideration, we define the following relation to represent interactions between sub-grammars (modules).

Definition 5 *Let $G'_1 = (N'_1, T'_1, P'_1, S'_1), G'_2 = (N'_2, T'_2, P'_2, S'_2)$ be two sub-grammars of grammar G, G'_1 calls G'_2, denoted as $G'_1 \gtrdot G'_2$, iff $vt_{S'_2} \in T'_1$, or in other words, iff there exists $A \in N'_1$ such that $A \rhd S'_2$.*

Now we can construct the *call graph* directly from the $\gtrdot$ relation to represent the control hierarchy between modules (sub-grammars) of a grammar. Clearly, the module containing the start symbol S is called by no modules and thus will appear at the highest level of the control hierarchy.

Let us take a sample grammar (Figure 2(a)) to illustrate the whole modularizing process. Figure 2(b) shows the dependency graph of nonterminals and Figure 2(c), 2(d) show the partition of nonterminals and the selection of start symbols respectively. The final results are given in Figure 2(e) and the call graph of modules is presented in Figure 2(f).

5. Testing

5.1. Module Testing

We now give the testing method which can be used for both module testing and the subsequent integration testing. In this section, we use the term *grammar* referring to both a complete grammar and its modules (sub-grammars).

Our testing method is derived from the two usages of grammars. Generally speaking, a grammar can be used in two ways: a generator which derives sentences and a recognizer which accepts valid sentences. To make the testing more adequate and precise, both aspects of a grammar should be considered. From this point of view, we propose two activities to accomplish the testing. One is to test the grammar with respect to a generator and the other test with respect to a recognizer.

- Test wrt a generator. Given a grammar that is being tested, generate a finite set of sentences which are then validated by the user, and, if not all sentences are accepted, modify the grammar and test again (Figure 3).

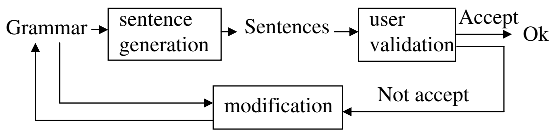


Figure 3. Test with respect to a generator.

- Test wrt a recognizer. The user is asked to give a few valid sentences of the intended language which is meant to be defined by the grammar under test, then sentence recognition is performed. If the recognition fails, which means that the grammar can not correctly recognize valid sentences, modifications are needed. If the sentences are accepted, then coverage analysis is done on the sentences by user-designated coverage criterion. If the coverage is satisfied, then the grammar being tested will pass the testing, otherwise some new sentences will be asked for and added to the initial set of sentences, then a new iteration starts (Figure 4).

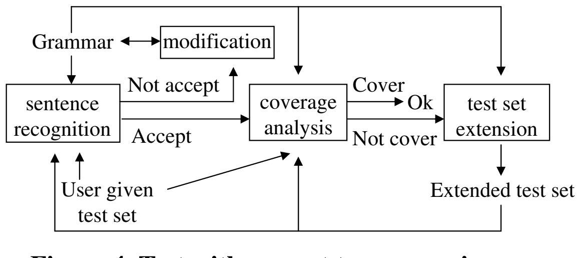


Figure 4. Test with respect to a recognizer.

The above two activities fit well with the two classes of faults with grammars (see Section 2.2). The first activity is expected to detect incorrectness of a grammar G by validating whether the generated sentence set (a subset of $\mathcal{L}(G)$) is a subset of the intended language L. The validation is totally done by user himself. On the other hand, in the second activity, the grammar G is used as a recognizer to decide whether the sentences given by the user (a subset of the intended language L) belong to $\mathcal{L}(G)$, which will detect incompleteness errors. Due to the interwinded character of incorrectness and incompleteness, the two activities should be performed interlacedly with each other.

By now, we have only considered using *positive* test cases (sentences) in the testing. Dually, we could also use *negative* test cases (non-sentences), say, generating non-sentences which should be refused by the user and asking user to provide invalid strings which should be rejected by sentence recognition. Non-sentences can be generated by applying mutation operators to the grammar [9].

Sentence generation, sentence recognition and coverage analysis are three main techniques involved in the testing process. Recognition can be conducted through sentence parsing. This is an old research topic and there are several approaches available such as Earley's algorithm [1]. Coverage criteria for grammars has been studied in [6] where a family of test criteria for CFGs is described and analyzed. There are also algorithms for automatic sentence generation such as Purdom's algorithm [12] and its extension [14]. We do not discuss deeply about the details of these techniques.

5.2. Integration Testing

In software testing, integration of modules can be accomplished either non-incrementally, testing each module in isolation and then combining them to form the program, or incrementally, combining the next module to be tested with the set of previously tested modules before it is tested [8]. For incremental integration, there are two strategies, top-down and bottom-up. The former starts with the top, or initial, module and then goes downwards to handle subordinate (called) modules, whereas the latter does exactly the opposite. These approaches can be directly applied to the integration testing of grammars. Particularly, one may prefer incremental integration as it is proved superior to non-incremental integration in software testing [8]. As for whether to follow top-down or bottom-up strategy, the decision may depend on the particular grammar under test.

An important aspect in each integration step is the transform from virtual terminal vt_A to nonterminal A once A is combined in. Referring to the example grammar in Figure 2 again, if we take the bottom-up integration, we first test modules G_3, G_4, then combine G_4 with G_2 where we need to change vt_F in G_2 to F, resulting in a new sub-grammar G_2' with start symbol A. After testing G_2', we combine it, plus G_3, with G_1, where all the existing virtual terminals are replaced by their corresponding nonterminals. This combination result is exactly the grammar in Figure 2(a) and then completes the whole integration testing.

6. Experimental Evaluation

We have implemented the framework and conducted experiments on two real-world grammars. In the implementation, we adopted Earley's algorithm [1] for sentence recognition, Purdom's algorithm [12] for sentence generation and Lämmel's context-dependent rule coverage [5] for coverage analysis. We have not implemented any algorithm for ambiguity detection but simply resorted to Bison by checking the shift/reduce and reduce/reduce conflicts reported by it. The C grammar and Java grammar used in our experiments are retrieved from the comp.compilers FTP at ftp://ftp.iecc.com/pub/file/.

The results of testing are summarized as follows. Both grammars are reduced but, as reported by Bison, are ambiguous. One ambiguity is known caused by the "if-then-else" clause. In dynamic testing phase, we found that the grammars are incorrect, in that they "over-specify" the languages, i.e., they admit constructs that are syntactically invalid. Incompleteness errors have not been detected yet. Below we first give the modularizing results, then describe some incorrectness errors found in the two grammars.

Table 1. Results of modularization.

Grammar			Modules				
	No. of		No. of modules			Avg No. of	
Name	nonterms	prds	total	scc	nonscc	nonterms	prds
C	64	213	10	3	7	6.4	21.3
Java	110	282	15	4	11	7.3	18.8

Table 1 presents the results of modularization. The three SCC modules for the C grammar correspond to the language components including *expressions* mixed with *declarations*, *statements* and *initializer*, while the non-SCCs correspond to *file* containing external-definition and function-definition, *unary-operator*, *assignment-operator*, etc. The call graph presented in Figure 5 reveals quite well the relations between these modules. The call graph for the Java grammar is omitted due to space limitation. In addition, as shown by Table 1, the number of SCCs (actually non-singleton SCCs of the dependency graph) is rather small, therefore our consideration of grouping singleton ones is quite necessary and turns out effective. Take the Java grammar for example, there are originally 67 SCCs with only 4 non-singleton and the remaining 63 singletons are then further grouped into 11 modules.

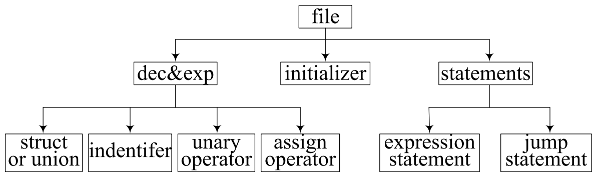


Figure 5. Call graph for the C grammar.

In the experiments, we simply followed non-incremental integration, testing each module independently then combining them to test the complete grammar. Two incorrectness errors found during this process are as follows.

The C Grammar. The error lies in that the grammar uses the same *declarator* to specify function declaration and variable declaration (see Figure 6). Actually, the declarators for functions and variables should be distinguished. This error was detected by the sentence "`indentifer{}`" generated at the last step of integration testing. This sentence is intended to define an empty C function but has syntax errors: the function name "`indentifer`" should be followed by "`()`" or "`(`parameters`)`". Since *declarator* and *function-definition* were partitioned into different modules, i.e., the former is in dec&exp module while the latter in file module (Figure 5), this error was not found until the modules were integrated.

```
 ··· ···
function_definition -> declarator function_body | ···
declarator -> declarator2 | pointer declarator2
declarator2 -> identifier | '(' declarator ')'
               | declarator2 '[' ']' | declarator2 '[' constant_expr ']'
               | declarator2 '(' ')' | '(' parameter_type_list ')'
               | declarator2 '(' parameter_identifier_list ')'
function_body -> compound_statement
compound_statement -> '{' '}' | '{' statement_list '}' | ···
 ··· ···
```

Figure 6. Segment containing incorrect productions of the C grammar.

The Java Grammar. The error is similar to that in the C grammar. The Java grammar does not distinguish the modifers "*public, protected, private*" from others like "*static, final*" (see Figure 7). This results in the following sentence containing an invalid class member declaration generated.

```
public class identifer{
   protected private byte identifer;
}
```

This error was detected during the testing of a module which describes the syntax for Java type declarations. It happens that the generation of the above sentence does not use any virtual terminals in that module.

```
 ··· ···
class_member_declaration -> field_declaration | method_declaration
field_declaration -> modifiers type variable_declarators ';' |···
modifiers -> modifier | modifiers modifier
modifier -> 'public' | 'protected' | 'private' | 'static' | 'abstract' | 'final' |···
type -> primitive_type | ···
primitive_type -> 'byte' | 'short' | ···
 ··· ···
```

Figure 7. Segment containing incorrect productions of the Java grammar.

Apparently, the "over-specify" can often make a grammar smaller and simpler, and thus easier to understand. The two grammars used in our experiments are likely written to be more readable than implementable. This probably explains why there are still errors detected by our framework.

In the process of testing, we found that it is a good idea to test a grammar from both aspects of its functionalities. On one hand, we noted that the user is more likely to provide sentences that are familiar or meaningful to him, which actually exploit only parts of the structure of the grammar. Sentences automatically generated however are quite different. They usually seem strange or unexpected to the user but are helpful for detecting errors. For example, the incorrectness errors mentioned above were detected exactly by generated sentences. This kind of error is not easily detected solely by user-provided sentences. On the other hand, we noted that since the generated sentences are relatively few (e.g., for the whole C grammar, only 11 sentences were generated) and, as mentioned above, are mostly *strange*, they alone could not test the grammar adequately. Therefore it would be better to combine with user-provided sentences.

7. Related Work

The notion of grammar testing was first formally discussed in [5] where the author developed concepts such as coverage analysis and test set generation for grammars, and discussed their application to grammar recovery. Harm and Lämmel [2] addressed the problem of testing attribute grammars, again focusing on coverage analysis and test generation. Li *et al.* [6] studied the formalization of test adequacy criterion for context-free grammars. They proposed a family of grammar test criteria and analyzed the subsume relations. Our work in this paper, different from the above, aims at devising sound and systematic testing methods for grammars from an engineering point of view.

Grammar modularization or grammar partition is largely discussed in the area of natural language processing. Wintner [16] defined formal semantics of context-free grammars and used it to derive a definition of modules and module composition in context-free grammars. Zajac and Amtrup [17] adopted the notion of modularity and composition and applied them to parser construction for unification-based grammars. Rather than partitioning automatically, they offered a pipeline-like composition operator that enables the grammar designer to break a grammar into sub-grammars. Weng *et al.* [15] also took the idea of partitioning a grammar into sub-grammars, each having its own parser, then combined to produce an overall one for parsing natural language queries. They gave some guidelines for partitioning a grammar based on its productions by utilizing the information provided by training data. Our partition method however is based on the nonterminals and partitions by utilizing the dependency relation between the nonterminals.

8. Conclusion

We have presented a systematic framework for grammar testing. We have proposed an approach to grammar modularization and a method for grammar module/integration testing. In the experiments on grammars for two programming languages, errors were found. We hope that the work outlined in this paper will make a contribution to the engineering discipline for grammars and also a complement to the traditional theory of software testing.

Our ongoing work includes experiments on more grammars and improvements on the current modularization method. Sentence generation for testing grammars is a particularly important and challenging problem on which we plan to take a deeper study in the future.

References

[1] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[2] J. Harm and R. Lämmel. Testing attribute grammars. In *Proceedings of the third workshop on attribute grammars and their applications (WAGA'00)*, pages 79–98, 2000.

[3] J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, November 1989.

[4] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (Second Edition)*. Addison-Wesley, Boston, MA, 2001.

[5] R. Lämmel. Grammar testing. In *Proceedings of Fundamental Approaches to Software Engineering (FASE'01) Vol.2029*, pages 201–216, 2001.

[6] H. Li, M. Jin, C. Liu, and Z. Gao. Test criteria for context-free grammars. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 300–305, 2004.

[7] M. Mernik, M. Crepinsek, and T. Kosar. Grammar-based systems: definition and examples. *Informatica*, 28(3):245–254, 2004.

[8] G. J. Myers. *The Art of Software Testing (Second Edition)*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

[9] J. Offutt, P. Ammann, and L. Liu. Mutation testing implements grammar-based testing. In *Proceedings of the Second Workshop on Mutation Analysis (Mutation'06)*, 2006.

[10] P.Klint, R.Lämmel, and C.Verhoef. Towards an engineering discipline for grammarware. *ACM Transaction on Software Engineering and Methodology*, 14(3):331–380, 2005.

[11] J. F. Power and B. A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution*, 16(6):405–426, November 2004.

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

[13] S. Schmitz. Conservative ambiguity detection in context-free grammars. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP'07)*, pages 692–703. Springer, 2007.

[14] Y. Shen and H. Chen. Sentence generation based on context-dependent rule coverag (in Chinese). *Computer Engineering and Applications*, 41(17):96–100, 2005.

[15] F. Weng, H. Meng, and P. C. Luk. Parsing a lattice with multiple grammars. In *Proceedings of the 6th International Workshop on Parsing Technologies (IWPT'00)*, 2000.

[16] S. Wintner. Modular context-free grammars. *Grammars*, 5(1):41–63, 2002.

[17] R. Zajac and J. W. Amtrup. Modular unification-based parsers. In *Proceedings of the 6th International Workshop on Parsing Technologies (IWPT'00)*, 2000.