Search-Based Ambiguity Detection in Context-Free Grammars

Naveneetha Vasudevan[1] and Laurence Tratt[2]

1 **Informatics, King's College London**
Strand, London, WC2R 2LS, United Kingdom. naveneetha@yahoo.com
2 **Informatics, King's College London**
Strand, London, WC2R 2LS, United Kingdom. laurie@tratt.net

Abstract

Context Free Grammars (CFGs) can be ambiguous, allowing inputs to be parsed in more than one way, something that is undesirable for uses such as programming languages. However, statically detecting ambiguity is undecidable. Though approximation techniques have had some success in uncovering ambiguity, they can struggle when the ambiguous subset of the grammar is large. In this paper, we describe a simple search-based technique which appears to have a better success rate in such cases.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases Ambiguity, Parsing

Digital Object Identifier 10.4230/OASIcs.ICCSW.2012.142

1 Introduction

Context Free Grammars (CFGs) are widely used for describing formal languages, including programming languages. The full class of CFGs includes ambiguous grammars—those which can parse inputs in more than one way. Since this causes conceptual and performance problems, most parsing algorithms can parse only a narrow subset of CFGs, avoiding ambiguity issues altogether. However, this is not without cost: the subsets are restrictive and rule out useful actions such as composing grammars. The starting point for this paper is that parsing using the full class of CFGs is a useful activity.

Ambiguity is a huge problem for machine processed languages, such as programming languages. If an input can be parsed in two ways, which should be taken? Unfortunately, we know that it is impossible to statically detect whether an arbitrary CFG is ambiguous or not [6].

Over the years, therefore, there has been a steady stream of work trying to uncover ambiguity in arbitrary CFGs. Exhaustive methods such as AMBER [9] systematically generate strings to uncover ambiguity, but even medium sized grammars quickly lead to unmanageable huge state spaces. Approximation techniques, on the other hand, sacrifice accuracy for termination. For instance, ACLA [5] is an approximation method where the original language of the grammar is extended into an approximated language that can be expressed with a regular grammar. Since all the strings from the original language are also included in the approximated one, there are no false negatives reported. However, the approximated language may contain strings that may not be part of the original one, and therefore the method can report false positives. Noncanonical Unambiguity (NU) Test is another approximation technique, where the original grammar is converted to a bracketed grammar by adding two terminals – a derivation (d_i) and a reduction (r_i), where i is the

2012 Imperial College Computing Student Workshop (ICCSW'12).
Editor: Andrew V. Jones; pp. 142–148

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

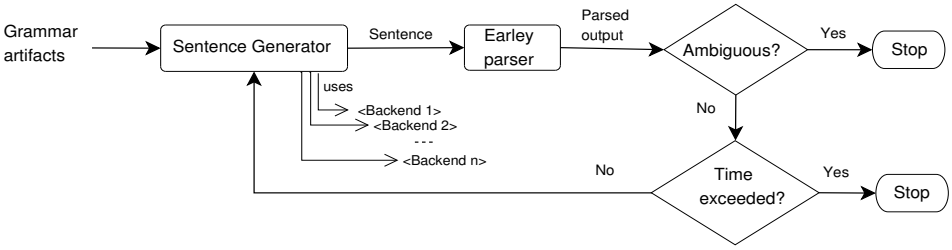

Figure 1 SinBAD architecture.

number of the production – at the front, and at the end of every grammar rule respectively. The introduction of these two terminals makes the bracketed grammar unambiguous. The challenge then, is to find two bracketed strings from the approximated grammar that map to a string in the original grammar. However, this method does not scale well for large grammars [3].

Hybrid approaches – where an approximation method is combined with an exhaustive method – increase the chances of detecting ambiguity. Basten's hybrid approach [4] – based on grammar filtering – applies an approximation method (NU Test) to filter out the unambiguous portions of the grammar, and then runs AMBER on the resulting smaller grammar to detect ambiguities. In principle, Basten's approach can be extended to other tools: ACLA, an approximation method, can be combined with CFG Analyzer [1], an exhaustive method, to search for ambiguous strings of bounded length. However, such hybrid approaches still rely on an exhaustive search although on a relatively smaller state space.

This paper is the first to explore a random search-based approach to grammar ambiguity detection. Given a grammar, our approach generates random strings, which are then parsed to detect ambiguity. In section 2.1 we describe our prototype tool: Search-Based Ambiguity Detection (SinBAD). In section 3 we set out the objective of our experiment, and then explain the choice of various data sets used for our experiment. In section 4 we compare and analyse our results. In section 5 we highlight the threat to validity of our random grammar generator, and finally in section 6 we conclude our experiment and provide future directions of our work.

2 Search-based ambiguity detection

Search-based techniques seek to find 'adequately' optimal solutions for problems that have no algorithmic solution and whose search space is too big to exhaustively scan. Such techniques have been applied to a wide range of problems including software itself (see e.g. [7]). Search-based techniques are either purely random or metaheuristic (such as hill climbing and genetic algorithms). Whereas in a random search the search space of candidate solutions is scanned randomly, in a metaheuristic search, a *fitness function* – to distinguish between a good and a poor solution – is used to guide the search. Since, this is the first paper to explore search-based techniques to ambiguity detection in CFGs, we have chosen the simplest search-based technique – a pure random search – for our experiment.

2.1 SinBAD framework

In this paper, we apply search-based techniques to ambiguity detection. We do so using a new tool, SinBAD, which allows us to experiment with different search-based approaches. Figure 1 shows SinBAD's architecture. Given a grammar and a lexer, the *Sentence Generator*

component generates random sentences using a *backend* instance. A backend, in essence, is an algorithm that governs how sentences are generated. For instance, a backend can use a unique scoring mechanism to favour an alternative when expanding a nonterminal, or one that can generate sentences of bounded length. The generated sentence is then fed to an Earley-based parser to check for ambiguity. The search stops when an ambiguity is found or when a time limit is exceeded. SinBAD can be downloaded from https://github.com/nvasudevan/sinbad.

2.2 Definition and Notations

A CFG is a four-tuple $\langle N,T,P,S\rangle$ where N is the set of nonterminals, T is the set of terminals, P is the set of production rules over $N \times N \cup T$ and S is the start symbol of the grammar. V is defined as $N \cup T$. A production rule $A: \alpha$ is denoted as $P[A]$ where $A \in N$, and α is V^*. We define a sentence of a grammar as a string over T^*. For a rule $P[A]$, $P[A]_{\text{alt}}$ denotes an alternative, and $\Sigma P[A]_{\text{alt}}$ denotes all its alternatives. The number of alternatives for a rule and the number of tokens in a rule are denoted as $\mathbb{N}(P[A])$ and $\mathbb{N}(P[A]_{\text{alt}})$ respectively. Notation $\mathbb{R}(L,n)$ indicates n items chosen randomly from a list L, and $\mathbb{R}[m..n]$ indicates a number chosen randomly between m and n.

2.3 Search-based backends

Given a grammar, Algorithm 1 describes how a sentence is generated. The function START is initialised with a grammar (G), the start time (t_s), the time duration (T) of search, and the threshold depth (D). To generate a sentence, we start deriving the start symbol S of the grammar by invoking the function GENERATE-SENTENCE recursively. To derive a nonterminal we randomly select one of its alternatives (line 11). We keep a note of when we have entered a rule and when we have exited. When the depth of the recursion exceeds a certain threshold depth, we start favouring alternatives (lines 8,9).

Algorithm 2 shows how an alternative is favoured for the Dynamic1 backend. When invoked for a rule, the function FAVOUR-ALTERNATIVE uses a scoring mechanism to favour an alternative. The score for an alternative is calculated as follows: terminal symbols are given a score of zero; for nonterminal symbols, the score is based on the ratio of their number of derivations that haven't been fully derived yet to the total number of derivations (line 8). One of the alternatives with a minimum score is then favoured.

3 Experiment

The objective of our experiment is to understand how well our search-based approach uncovers ambiguity. Since ambiguity is inherently undecidable, it is impossible to evaluate such a tool in an absolute sense. Instead, we evaluate our approach against two other tools – ACLA and AmbiDexter [2] – and on two sets of grammars: 1000 grammars that we have randomly generated[1]; grammars for Pascal, SQL, Java and C that have been manually altered to be ambiguous[2].

The three tools differ in their approach: ACLA uses an approximation technique; AmbiDexter uses a hybrid approach; and SinBAD uses a search-based approach. We evaluate these three tools for both sets of grammars for varying time limits – 10, 30, 60, and 90 seconds – to understand how long each tool takes to uncover reasonable quality results. For

[1] Available at https://github.com/nvasudevan/sinbad/tree/master/experiment.
[2] Taken directly from [4].

Algorithm 1 Algorithm for generating a sentence

```
1: function START(G, t_s, T, D)
2:     return GENERATE-SENTENCE(P[S], G, t_s, T, d = 0, D)
3: end function
4: function GENERATE-SENTENCE(P[A], G, t_s, T, d, D)
5:     exit if time_elapsed(t_s, T)
6:     Sen ← empty string
7:     P[A].entered ← P[A].entered + 1                    ▷ We enter rule
8:     if d ≥ D then
9:         P[A]_alt ← FAVOUR-ALTERNATIVE(P[A], G)
10:    else
11:        P[A]_alt ← ℝ(ΣP[A]_alt, 1)
12:    end if
13:    for each V ∈ P[A]_alt do
14:        if V ∈ N then
15:            Sen ← Sen + GENERATE-SENTENCE(P[V], G, t, T, d + 1, D)
16:        else
17:            Sen ← Sen + V
18:        end if
19:    end for
20:    P[A].exited ← P[A].exited + 1                      ▷ We exit rule
21:    d ← d − 1
22:    return Sen
23: end function
```

Algorithm 2 Algorithm for favouring an alternative for Dynamic1 backend

```
1: function FAVOUR-ALTERNATIVE(P[A], G)
2:     scores ← [ ]
3:     for each P[A]_alt ∈ ΣP[A]_alt do
4:         score_alt ← 0
5:         for each V ∈ P[A]_alt do
6:             if V ∈ N then
7:                 if P[V].entered > 0 then
8:                     score_alt ← score_alt + (1 − (P[V].exited/P[V].entered))
9:                 end if
10:            end if
11:        end for
12:        scores ← score_alt
13:    end for
14:    alts_min ← { alt | ∀alt ∈ ΣP[A] ∧ score_alt = min(scores) }
15:    return ℝ(alts_min, 1)
16: end function
```

the (generally much larger) programming language grammars, we also evaluate the tools for extended periods (180 and 300 seconds) as the number of production rules is much higher than for our random grammars.

We evaluate AmbiDexter for two versions of a grammar—unfiltered and filtered (with

SLR1). AmbiDexter provides an option for generating filtered versions of a grammar. For random grammars, we generate the filtered version, and for the altered programming language grammars, we take it directly from [4]. We evaluate SinBAD with the Dynamic1 and Dynamic2 backends for two threshold depths (D), 10 and 30. We have chosen these two values for depth to uncover reasonably long ambiguous fragments. Our experiment was performed on an Intel Core2 Quad Q9450 2.66GHz machine with 4 GB of memory. The maximum JVM heap size for ACLA and AmbiDexter was 2048Mb.

3.1 Random grammar generation algorithm

Algorithm 3 outlines the algorithm for our random grammar generator. We initialise nonterminal and terminal sets with equal numbers of symbols. To generate an alternative, a token is picked randomly from set V. Each rule can have 1 or more alternatives, and each alternative can have 0 or more symbols. The maximum number of alternatives for a rule and the maximum number of tokens in an alternative is controlled by the MAX_{alts} and $\text{MAX}_{\text{tokens}}$ parameters respectively. The MAX_{ϵ} controls the maximum number of empty alternatives.

Algorithm 3 An algorithm for generating a random grammar

```
1:  function GENERATE-GRAMMAR(MAX_alts, MAX_tokens, MAX_ε)
2:      P ← { }
3:      N ← Set of nonterminals
4:      T ← Set of terminals
5:      V ← N ∪ T
6:      N_ε ← ℝ(N, MAX_ε)
7:      for each A ∈ (N ∪ S) do
8:          N_alts ← ℝ[1..MAX_alts]
9:          while ℕ(P[A]) < N_alts do
10:             P[A]_alt ← [ ]
11:             N_tokens ← ℝ[1..MAX_tokens]
12:             while ℕ(P[A]_alt) < N_tokens do
13:                 P[A]_alt ← P[A]_alt + ℝ(V, 1)
14:             end while
15:         end while
16:         P[A] ← P[A] + [ ] if A ∈ N_ε          ▷ Append an empty list
17:     end for
18:     return ⟨N,T,P,S⟩
19: end function
```

All the grammars the algorithm generates are syntactically valid, though there is no guarantee that they resemble 'real-world' grammars. For example: a grammar with a start rule `S: x` can't be derived further; a rule `A: A` with no other alternatives never terminates.

4 Comparison and Analysis

Table 1 displays the results of our experiment. We now present a brief analysis of some of the most interesting parts.

Given a grammar, ACLA will report it to be ambiguous, unambiguous, or possibly ambiguous (that is, it is unsure if the grammar is ambiguous). For both sets of grammars,

Table 1 Number of ambiguities detected for random and programming language grammars.

		ACLA	AmbiDexter		SinBAD			
	Time	-	-		Dynamic1		Dynamic2	
	(seconds)		Unfiltered	SLR1	D=10	D=30	D=10	D=30
Random CFGs	10	81	355	356	357	15	499	26
	30	201	373	371	499	57	634	55
	60	316	376	371	545	54	631	80
	90	360	378	376	554	72	629	82
Altered real-world CFGs	10	14^{bc}	16^{ab}	16^{ab}	20	18^{b}	16^{ac}	17^{ab}
	30	14^{bc}	16^{ab}	16^{ab}	20	18^{b}	16^{ac}	18^{ab}
	60	14^{bc}	16^{ab}	16^{ab}	20	18^{b}	16^{ac}	18^{ab}
	90	14^{bc}	16^{ab}	16^{ab}	20	19^{a}	16^{ac}	19^{a}
	180	15^{bc}	18^{ab}	19^{b}	20	20	16^{ac}	19^{a}
	300	15^{bc}	18^{ab}	19^{b}	20	20	16^{ac}	20

a) Ambiguity not found for at least one of: Java.1, Java.3, and Java.4
b) Ambiguity not found for at least one of: C.1, C.2, C.4, C.5
c) Ambiguity not found for at least one of: Pascal.3, Pascal.5

ACLA performs better when we increase the time limit. For random grammars, ACLA did not report any grammar to be unambiguous. For the altered programming language grammars, Pascal.3 and Pascal.5 were reported to be possibly ambiguous. Analysis for the (large) C grammars – C.1, C.2 and C.4 – did not complete within a time limit of 300 seconds.

AmbiDexter fared better than ACLA for both sets of grammars. For random grammars, increasing the time limit does not lead to a significant increase in the number of ambiguities found. This is because AmbiDexter searches for ambiguity based on increasing sentence length. Therefore, for grammars with a short ambiguous fragment, AmbiDexter is quick to find it. However, when the ambiguous fragment is long, AmbiDexter struggles. For the altered programming language grammars, the results were slightly better for the filtered version set. This is because in filtered grammars, production rules that do not contribute to ambiguity are filtered out, thus resulting in a smaller state space. Further, we noted that for larger grammars (such as C), increasing the time limit lead to better results.

SinBAD, for random grammars, performs better for a lower value of threshold depth (D=10) than for a higher value (D=30). This is because, for case D=10, sentence generation is quick whereas for case D=30, sentence generation takes much longer. Generating sentences quicker allows the search to try a greater number of sentences possible, thereby increasing the chances of detecting ambiguity. Further, Dynamic2 – which has a better mechanism to converge sentence generation than Dynamic1 – performs better. For the altered programming language grammars, Dynamic1 performs better than Dynamic2. Dynamic1 uses a scoring mechanism that ensures every alternative gets an opportunity to be selected for sentence generation. Dynamic2, however, uses a scoring mechanism that focuses on converging the sentence generation. As a result, Dynamic1 covers a much wider area of the search space than Dynamic2. As table 1 shows, SinBAD performs much better on random grammars than the other tools, and performs at least as well on altered programming language grammars.

We also noted that whilst the number of ambiguities found for ACLA, AmbiDexter, and SinBAD's Dynamic1 stayed the same or increased, Dynamic2 got slightly worse with increased time limits and D=30. This is because both ACLA and AmbiDexter search through the state space systematically, and therefore the search space for higher time limits is inclusive of the search space for lower time limits. SinBAD, however, randomly selects points in the search space, and can give substantially different results from run to run.

5 Threats to validity

The most obvious threat to validity is our random grammar generator. We have no easy way of being confident that the CFGs it produces span the entire possible set of CFGs. Although we wrote the generator without any particular ambiguity tool in mind, it may produce a subset of CFGs which unintentionally favour SinBAD's algorithms. In the future, we hope that a CFG equivalent of the work on random generation of automata [8] may be developed. By using Basten's set of manually altered real programming language grammars, we have some confidence that SinBAD's algorithms work well beyond our random grammars.

6 Conclusions

In this paper, we introduced the concept of a search-based approach to CFG ambiguity detection. Our experiments show that simple techniques give promising results, detecting a larger number of ambiguities in random grammars than previous tools, and executing in reasonable time. Our next step is to add more tools to the study and perform a larger experiment with more real-world-esque grammars to see if these initial results apply to the sort of CFGs that tend be to be used in practice.

References

1 Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental sat solver. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 410–422. Springer-Verlag, 2008.

2 Bas Basten and Tijs van der Storm. Ambidexter: Practical ambiguity detection. In *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010*, pages 101–102. IEEE Computer Society, 2010.

3 H.J.S. Basten. Msc. thesis. Master's thesis, 2007.

4 H.J.S. Basten and J. J. Vinju. Faster ambiguity detection by grammar filtering. In *Proc. of the Tenth Workshop on Language Descriptions, Tools and Applications*, pages 5:1–5:9. ACM, 2010.

5 Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, March 2010.

6 David G. Cantor. On the ambiguity problem of backus systems. page 477–479, 1962.

7 Mark Harman. The current state and future of search based software engineering. In *FOSE*, pages 342–357, 2007.

8 Pierre-Cyrille Héam, Cyril Nicaud, and Sylvain Schmitz. Random generation of deterministic tree (walking) automata. In *Proceedings of the 14th International Conference on Implementation and Application of Automata (CIAA'09)*, volume 5642 of *Lecture Notes in Computer Science*, pages 115–124. Springer-Verlag, July 2009.

9 Friedrich Wilhelm Schröer. Amber, an ambiguity checker for context-free grammars. Technical report, 2001. `http://accent.compilertools.net/Amber.html`.