Ambiguity detection for programming language grammars
Basten, H.J.S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 6

Implementing AMBIDEXTER

“I’d give my right arm to be ambidextrous.”
Brian W. Kernighan

This chapter presents our tool implementation of AMBIDEXTER. The tool was
developed to experimentally validate the techniques presented throughout this thesis,
but also to be used in real grammar development. The tool consists of two parts: a
harmless production rule filter, and a parallel sentence generator. We discuss the
architecture as well as implementation details of both of these parts, and finish with
advise for their usage.

6.1 Introduction

To be able to validate our grammar filtering techniques described in Chapters 3–5 on real
programming language grammars, we needed to implement them in a tool. Furthermore, to
test the effect of filtering on exhaustive searching techniques we also needed implementations
of these methods. For validating the filtering of token-based grammars we could use existing
tools like AMBER and CFG ANALYZER, but for character-level grammars there was no tool
available. We therefore created our own sentence generator for this type of grammars, and
combined it with our grammar filter into a single tool called AMBIDEXTER. In this chapter
we discuss the implementations of both parts of the tool.

This chapter is based on a tool demonstration paper with the same title that was published in the proceedings
of the Tenth IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010) [BvdS10]. This
paper was written together with Tijs van der Storm.
6.2 Grammar Filter

The first part of the tool implements the grammar filtering technique described in Chapters 3, 4 and 5. In this section we assume a basic understanding of the grammar filtering process. For a detailed introduction we refer to Sections 5.2 and 5.4.

6.2.1 Requirements

The main functional requirement of the tool is of course that it should correctly implement our grammar filtering technique and its character-level extensions. But apart from that, it should also fulfill the following non-functional requirements:

- **Memory efficient** Because the number of item pairs can be very high, the pair graph traversal is much more memory intensive than it is cpu-intensive.
- **Extensible** To allow for easy experimenting with new techniques or improvements.
- **Explanatory** The tool should provide insight into the applied techniques, and might serve as a reference implementation.

In the next section we describe the architecture of the grammar filtering tool, and describe how it meets these requirements.

6.2.2 Architecture and Design

Figure 6.1 shows an overview of the architecture of the grammar filtering tool. The process starts from an input grammar and a precision setting for the approximation. The input grammar can be read in YACC [Joh]/BISON [DS05], SDF2 [HHKR89, Vis97] or RASCAL [KvdSV11] format. A nondeterministic finite automaton (NFA) is build from the grammar that approximates its parse trees with the specified precision. The states of this NFA are the items of the grammar and are, depending on the chosen precision, extended with lookahead information. Our tool supports the following precisions: LR(0), SLR(1), LALR(1) and LR(1). The precisions are named after parsing algorithms, because their resulting NFAs resemble their corresponding nondeterministic parse automata.

After the basic NFA is constructed, several precision improving operations are made to it, depending on the type of grammar or options selected by the user. Special **NFA modifiers** can be loaded that each alter the nodes and edges of the NFA, for instance by removing edges or unfolding certain paths. The priority and associativity annotations of character-level grammars as described by Algorithm 7 in Section 5.5.2 are implemented like this, as well as the follow restriction propagation of Algorithm 8. To satisfy the extensibility requirement, new modifiers can be added modularly.

After the NFA is extended, we filter out the states that are not used in the description of parse trees of potentially ambiguous strings. To find these states we build the pair graph. The nodes of the pair graph are pairs of NFA nodes, extended with some additional information. A path through this graph marks two paths through the NFA that describe different parse trees for the same potentially ambiguous string. Therefore, the nodes that need to be filtered from
the NFA are those that are not used on any complete path through the pair graph. The pair graph is traversed while it is constructed and the used NFA states are collected — as described in Algorithm 4. After that, the unused states are filtered from the NFA, and loose ends are pruned.

The process of pair graph traversal and NFA filtering is repeated until no more states can be filtered. After that, the NFA contains only those states that contribute to possible ambiguity under the current approximation precision. At this point, this information can be used in
two ways. The first one is to collect the potentially harmful production rules from the NFA and reconstruct a new grammar from them. This grammar can then be searched further with other ambiguity detection methods. Output formats for the following tools are supported: YACC/BISON, ACCENT/AMBER [Sch06, Sch01], CFG ANALYZER [AHL08] and the implementation of the “Ambiguity Checking with Language Approximation” framework [BGM10]. The other option is to reconstruct unproductive states in the NFA and store it to disk, so that it can be used with AMBIDEXTER’s sentence generator immediately.

To support modifications to the pair graph while keeping a single traversal algorithm, special pair graph extensions can be loaded. An extension consists of a set of functions that are called during the traversal of each item pair. The functions are able to store additional information in an item pair, and can use this information to abort the traversal at certain points. As an example, the handling of reject filters in character-level grammars — as described in Section 5.5.2 — is implemented as a pair graph extension. The reject extension stores two additional sets of reduced non-terminals with each item pair, and aborts the traversal of an item pair if it encounters the reduction of a rejected non-terminal.

Another important component in the architecture of the pair graph traversal is the item pair store. The purpose of the item pair store is to keep track of all created item pairs and their traversal related information in a memory efficient way. Because the number of item pairs is quadratic in the number of NFA states, pair graphs can become very large. As Table 6.1 shows, the pair graphs required for checking real programming language grammars can grow up to hundreds of millions of pairs. Without efficient storage, they would be impossible to traverse on modern machines. The following section discusses the current implementation of the item pair store in more detail.

6.2.3 Implementation Details

The biggest problem with creating a practically usable grammar filtering tool was to find a memory efficient pair graph representation. For checking character-level grammars, we need to record at least the following information per item pair:

- Two NFA states
- Two conflict flags
- number and lowlink (see Algorithm 4)
- alive and on stack flags (see Algorithm 4)

Furthermore, we need to store all created item pairs in a data structure to be able to test whether a newly visited item pair is already visited before or is new. In the first prototype implementation we used plain Java objects to represent an item pair, together with a simple hashmap to store the created objects. For the experiments on token-based grammars shown in Chapter 3 this design performed sufficiently well. However, for running the character-level grammars experiments of Chapter 5, it turned out to be too memory inefficient.

In order to take the reject filters into account during the pair graph traversal, we needed to add two additional sets of reduced non-terminals to our item pairs (see Section 5.5.2). Together
with the information mentioned above, our item pair class now consisted of 7 fields. On a 64-bit Java virtual machine, the object size for such a class is 80 bytes of memory. Furthermore, each object required an additional 32 bytes for a bucket entry object in the hashmap. Running the CHAR+UNF experiment on the Java grammar shown in Table 6.1 would therefore take up at least 33GB of memory. However, this number gets even larger if we also take into account the space required for storing the loaded grammar and the NFA. Our initial implementation was not efficient enough for checking character-level grammars, so we needed a more compact item pair representation.

We looked for inspiration in the closely related field of model checking, where checkers also need to explore large state spaces. The first idea we adopted was to use bit strings to represent item pairs. In the current implementation we use one 64-bit Java long to represent a pair’s identity (NFA states, flags and extension information), and another long for the traversal related information (number, lowlink, alive and on stack). The first long contains two stretches of bits to represent an NFA state pair, of which the size depends on the total number of states, and two bits for the conflict flags. The remainder of the bits can be reserved by the used item pair extensions to store additional information. The traversal related bit string contains two 31-bit fields to represent a pair’s number and lowlink, and two single bit flags for alive and on stack. The maximum number of item pairs that can be created with this solution is therefore limited to $2^{31}$, but this is more than enough for checking the programming language grammars.
Chapter 6. Implementing AMBIDEXTER

To limit the memory overhead of storing the already traversed item pairs we used a custom hashmap. Instead of using linked lists for its buckets like the java.util.HashMap does, it uses plain arrays to store the bit strings. This requires only a small logarithmic space overhead, instead of the constant 32 bytes for linked list elements. The arrays are searched for item pairs linearly, which does not result in noticeable speed losses if they are not that long. Therefore, the hashmap is re-hashed when needed to make sure the arrays do not grow beyond a certain size — which is currently 2048. This way, the item pair store makes efficient use of heap space without large performance penalties.

We also considered implementing other techniques used in model checking. For instance, we experimented with storing the bit strings in BDDs [Ake78]. However, the typical distribution of the bit strings and their relatively short lengths seem to be unsuitable for reaching good compression rates. Other space saving options would be to store chunks of bit strings to disk or compress them in memory. However, this will probably result in a loss of speed. We therefore chose for the array-backed item pair store described above, because it performed sufficiently well for the real world grammars tested in Chapter 5, as can be seen from Table 5.2.

6.3 Sentence Generator

In order to validate the effect of our character-level grammar filter, we also needed an exhaustive detection method for character-level grammars. We chose for a simple depth-first sentence generation technique, similar to that of AMBER. This section highlights the design decisions we made while developing the sentence generation tool.

6.3.1 Requirements

The main functional requirement for our exhaustive ambiguity detection tool was that it should find ambiguous sentences in character-level grammars as well as token-based grammars. Together with their parse trees, ambiguous sentences are the most descriptive proofs of the ambiguity of a grammar. To not report ambiguities that are already solved by the grammar developer, the tool should take disambiguation filters into account.

The main non-functional requirement of the exhaustive searching tool was that it should be as fast as possible. The higher the number of sentences that can be explored per minute, the higher the chance that ambiguities will be found. Furthermore, we wanted our tool to be suitable for short interactive use, as well as longer overnight runs.

6.3.2 Architecture and Design

To meet the above requirements we chose to implement a simple depth-first sentence generation technique, similar to that of AMBER. AMBER uses a modified Earley parser that generates sentences on the fly while parsing them. In our case however, we chose to use an SGLR parser, because knowledge about applying disambiguation filters in this parsing technique was readily available [vdBSV02, Vis97]. Furthermore, LR parsers are generally faster than Earley parsers because they use a precomputed parse table.
Our second design decision was to generate sentences in a depth-first fashion. This requires only very little memory, whereas with breadth-first searching the memory usage grows with the number of sentences produced. From the results presented in Section 3.3.2, we see that the breadth-first CFG ANALYZER already used 1.3GB of memory after searching for 106 minutes. We therefore expect depth-first search to be a better candidate for running longer checks, even more so because the languages of character-level grammars are typically much larger than
the token-based grammars explored in Section 3.3.2. Furthermore, depth-first searching is much easier to parallelize than breadth-first searching. We can therefore take advantage of present-day multicore processors for even better performance.

Figure 6.2 shows the architecture of the sentence generator. The sentence generation process can start either from a — possibly filtered — grammar or an NFA that was reconstructed after filtering. In case a grammar is given, its LR(0) NFA is generated from it, which in turn is converted into a deterministic LR(0) pushdown automaton. A reconstructed NFA is converted to deterministic form immediately.

The pushdown automaton is then used to generate sentences of a predefined length $k$. This can be done by a certain number of sentence generators in parallel. First, a single sentence generator is used to generate all prefixes of a given length $l < k$ which are stored in a set. After that, the parallel sentence generators take the prefixes from this set and generate all their completions up to length $k$. All ambiguous sentences that they find are parsed with a small parser and are reported to the user, together with their parse trees. These parse trees can then be searched for the causes of ambiguity in the grammar, either by hand, or with an expert system like DR. AMBIGUITY (see Chapter 7).

6.3.3 Implementation Details

The sentence generator is implemented as a normal SGLR parser (see [Vis97], Chapter 3), with the following modifications:

- After all stacks at a certain level are reduced, a set of shiftable characters is calculated.
- For each level such a set of candidate characters is stored, using an additional stack.
- The characters to shift are picked and removed from these sets, instead of read from an input string.
- If the maximum string length is reached, or if the set of candidates at a certain level is empty, all stacks are backtracked one level, and a new shift is tried.
- The garbage collection only removes stack nodes that are popped during backtracking.
- Follow restrictions do not have to be checked, because they are already propagated through the parse automaton before determinisation — see Algorithm 8 in Section 5.5.2.
- For speed, parse trees are not built during generation, but are obtained by re-parsing ambiguous strings.

These changes were relatively easy to implement, except for the selection of the candidate characters to shift at each level. Especially with character-level grammars, just gathering all characters that can be shifted from each stack can lead to an unnecessarily high number of generated sentences. For instance, consider the typical lexical definition $[a-zA-Z]+ [a-zA-Z0-9_]$ for an identifier. This definition generates a very high number of possible identifiers. However, these do not all have to be explored if they will never lead to ambiguities.
Algorithm 10 Finding an approximation for the smallest set of characters that together continue the current gss with all possible combinations of paths.

function GET-SHIFTABLE-CHARACTERS(top ∈ \(\mathcal{P}(Q)\)) =
1  \(\text{// gather all shift actions of states in top}\)
2  \(\mathcal{A} = \{ q \rightarrow a \rightarrow q' \in \text{shift} \mid q \in \text{top} \}\)
3  \(\text{// find a set } \mathcal{B} \subseteq T \text{ s.t. } \forall q_1 \rightarrow a \rightarrow q_1', q_2 \rightarrow a \rightarrow q_2' \in \mathcal{A} : \exists b \in \mathcal{B} : q_1 \rightarrow b \rightarrow q_1', q_2 \rightarrow b \rightarrow q_2' \in \mathcal{A} \text{. }\)
4  \(\text{// preferably the smallest}\)
5  \(\mathcal{B} = \emptyset\)
6  while \(|\mathcal{A}| > 0\) do
7      pick a \(b \in T\) that occurs the most often in \(\mathcal{A}\)
8      remove the elements \(q \rightarrow a \rightarrow q'\) from \(\mathcal{A}\) for which hold that \(q \rightarrow b \rightarrow q'\)
9      add \(b\) to \(\mathcal{B}\)
10  od
11  return \(\mathcal{B}\)

An ambiguity only appears in case two stacks merge upon reduction of the same substring. Therefore, it suffices to select only enough candidate characters such that every possible combination of stacks will be explored, and of course every individual stack as well. This will lead to the fastest detection of existing ambiguities.

Selecting this minimum set of shiftable characters to cover all combinations of stack continuations is an instance of the hitting set problem [Kar72], which is NP-complete. In our current implementation we use a simple non-optimal computation shown in Algorithm 10. As of yet, it has never led to noticeable performance losses, because the number of stacks and possible shift transitions are usually quite low. More experiments are required to test whether finding the smallest possible set of candidates will pay off.

6.4 Usage

To get the fastest results, we advise the following strategy for finding ambiguities with AMBIDEXTER:

When the grammar under investigation contains a lot of ambiguous production rules, their ambiguity will propagate to other production rules as well during the approximative search for harmless productions. This reduces the chance of actual harmless rules being found. Therefore, we advise to quickly test for ambiguities with the sentence generator first. If ambiguities pop-up during this search, they can be solved first before trying the harmless rule filter.

When there are no 'low hanging' ambiguities to be found anymore, the sentence generation can be sped up by filtering harmless productions from the grammar. Depending on the size and shape of the grammar, the higher filtering precisions like LR(1) or grammar unfolding might require very large amounts of memory. Therefore, its best to start checking with lower precision settings first, and then gradually increase until a configuration is found that runs within acceptable time and memory limits. Judging from the figures in Table 5.2, this process
will probably not take more than a couple of minutes for small to medium sized grammars.

If the grammar filter finds harmless production rules, the filtered grammar can be tested again with the sentence generator. Hopefully the sentence generation will now be faster so it can search for ambiguities at longer sentence lengths. If more ambiguities are found they can be removed from the grammar. Then, the grammar filter can be run again to see if more production rules are harmless. This process of filtering and sentence generation can be continued until it becomes unfeasible, or until the grammar filter finds the grammar to be unambiguous.
6.5 Conclusion

In this chapter we have presented our tool implementation of AMBIDEXTER. The tool implements our grammar filtering techniques, as well as a sentence generator for token-based grammars and character-level grammars. We have discussed the architectural design and implementation details of these two parts, together with advice on how they can be used together in an optimal way.

The AMBIDEXTER tool is integrated in the RASCAL IDE in Eclipse. Figure 6.3 shows the wizard that can be used to configure it. For checking YACC or SDF2 grammars AMBIDEXTER can be run from the command-line as well. To get an indication of the development effort invested in the tool, Figure 6.2 shows the sizes of the AMBIDEXTER’s packages measured in lines of code.

<table>
<thead>
<tr>
<th>Package</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar (incl. import/export)</td>
<td>3958</td>
</tr>
<tr>
<td>Automata</td>
<td>3408</td>
</tr>
<tr>
<td>Pair graph traversal</td>
<td>1817</td>
</tr>
<tr>
<td>Sentence generation</td>
<td>1009</td>
</tr>
<tr>
<td>Parser</td>
<td>888</td>
</tr>
<tr>
<td>Utilities</td>
<td>2039</td>
</tr>
<tr>
<td>Tests</td>
<td>563</td>
</tr>
<tr>
<td>Main</td>
<td>492</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td>14180</td>
</tr>
</tbody>
</table>

Table 6.2: Sizes of AMBIDEXTER’s packages in non-comment source lines of Java code (SLOC). Figures generated using David A. Wheeler’s ‘SLOCCount’.