Natural Language Parsing for Fact Extraction from Source Code

Jens Nilsson∗ Welf Löwe∗ Johan Hall† Joakim Nivre†∗
∗Växjö University, School of Mathematics and Systems Engineering, Sweden
†Uppsala University, Dept. of Linguistics and Philology, Sweden
{jens.nilsson|welf.lowe|johan.hall|joakim.nivre}@vxu.se

Abstract

We present a novel approach to extract structural information from source code using state-of-the-art parser technologies for natural languages. The parser technology is robust in the sense that it guarantees to produce some output, entailing that even incomplete or incorrect source code as input will get some kind of analysis. This comes at the expense of possibly assigning a partially incorrect analysis for input free of errors. However, an evaluation on source codes of the Java, Python and C/C++ languages shows that the committed errors are few i.e., our accuracy is close to 100%. The error analysis indicates that the majority of the errors remaining are harmless.

1. Introduction

The first step in reverse-engineering is usually information extraction. As long as the documents containing the desired information (e.g., the source code of programs) adhere to a formal language (e.g., a programming language), classic analysis techniques known from the field of compiler construction may be applied to construct formal models (e.g., abstract syntax trees). This, however, is not always the case. First, the documents may be incomplete since they are under development; still one wants to have development support from a reverse-engineering tool, e.g., checking the conformance of the design documents with the developed program. Secondly, the documents may be erroneous; reverse-engineering tool support is then required even more so. Thirdly, the documents may adhere to a dialect of the formal language that has evolved in a company or a special domain. This dialect is not understood by the standard information extractor of the reverse-engineering tool. In all these cases, front-ends in reverse-engineering tools loose a lot of information or simply break. The novel approach presented here applies natural language parsing in order to produce syntax trees under these more difficult circumstances.

Software analysis and reverse-engineering usually have low priority and we face a lack of resources for measurement and improvement – except in emergencies when we observe the need for immediate results [13]. If this is true, there is little chance to eliminate the aforementioned problems by putting a lot of effort in careful front-end designs covering even incomplete and erroneous documents of the source language and their dialects. Instead, adapting the information extraction to the given information sources is on the critical path from problem symptom detection to problem understanding (supported by reverse-engineering tools) and problem solving.

We observe three objectives for information extractors: (i) they obviously need to be robust, i.e., they should always give a meaningful model even for slightly incorrect and incomplete input. Not quite as obvious, (ii) they ought to be developed rapidly for new languages and dialects. Finally, (iii) they ought to be accurate, i.e., they should give the correct analysis result for a correct source document. However, due to further abstraction of the source information and the fuzzy nature of many reverse-engineering problems, 100% accuracy is dispensable.

The rapid development of robust information extractors is of special interest for languages like C/C++ due their numerous dialects in use [1]. Programmers using a special dialect, who want to perform a program analysis on their codes, may accept approximated analysis models and can live with the fact that analysis is not 100% accurate. The rapid development of robust fact extractors can also be useful in analyzing new version of a programming language such as for Java. An existing fact extractor for older versions based on a grammar is unusable for the new version. Usually it requires a large amount of programming or specification labor to adapt to the new version. Existing robust fact extractors for programming languages only work for small sets of languages. Still, it requires a lot of manual work to port them to other languages.

The natural language processing (NLP) community has for many years developed information extraction technology that is both highly accurate and completely robust. Ro-
Bustness is required since an exact formal description of natural language is hard to define (if such a description can exist at all). In NLP, fact extractors for various natural languages and dialects can be constructed quite rapidly. This approach only needs correct examples of the source and the expected analysis model. Then it automatically trains and adapts a generic parser. It is, hence, less time consuming to adapt to a new language provided such training data are available. As we will show, training data for adapting to a new programming language can even be generated automatically.

In this paper, we adopt natural language parsing to information extraction from source code of programming languages. The paper contributes with: (1) a methodology to 100% robust and highly accurate information extraction from source codes, efficiently adapting to new languages, and (2) experimental results for C/C++, Java, and Python supporting our claims.

Section 2 discusses related NLP work and describes the preparation of the training examples necessary for the NLP parser applied here, while section 3 presents the experiment results. Section 4 discusses related work in information extraction for reverse-engineering. We end with conclusions and future work in section 5.

2 General Approach

The syntactic structure of formal languages, e.g., programming languages, is defined using context-free grammar and, for each program, captured in and abstract syntax tree (AST) containing both terminals and nonterminals. This is also the standard for natural languages. Dependency structure is another way of representing the syntax of natural languages [17]. Dependency trees form labeled, directed and rooted trees. One essential difference compared to context-free grammar is the absence of nonterminals.

The NLP parser producing such dependency trees, used in the experiments, is MaltParser (maltparser.org). It is in many ways similar to the shift-reduce parser for context-free grammars. The construction of syntactic structure is created by a sequence of transitions, with the most apparent difference that terminals are pushed onto and popped from the stack but never reduced to nonterminals. Parsing has a time complexity linear in the length of the input [17].

In contrast to a parser guided by a grammar (e.g. ordinary shift-reduce parsing for context-free grammars), this parser is guided by a machine learning classifier [18]. Hence, the parser requires benchmark data (also referred to as training data) containing sentences and corresponding dependency trees. In other words, the parser has a training phase where the training data is used by the training module in order to learn the correct sequence of transitions. The training data contains dependency trees for sentences of any natural or formal language, entailing that the parsing algorithm is language-independent.

One consequence of guiding the parser using a classifier – compared to using a grammar – is that it guarantees that some kind of syntactic analysis will be produced even though the input does not conform to a grammar. The price we have to pay for this robustness is that any classifier can commit errors even if the input is acceptable according to a grammar.

We need to adapt natural language parsing to the needs of information extraction from programming language codes. Our general approach can be divided into two phases: (i) training and (ii) production. The general strategy is shown in figure 1.

Phase (i) first generates training data by automatically producing first syntax trees and then dependency graphs for correct programs. Then we train the generic parser with the training data which is actually identical to the training for parsing natural languages. Once the generator for the training data is adapted to a programming language, phase (i) is performed fully automatically. It needs to be done for once of every new programming language to adopt. We have done it for Java, Python and C/C++.

Phase (ii) extracts the information from (not necessarily correct and complete) programs. Therefore, it first parses the source code into dependency graphs. As the back-ends of reverse engineering tools rely on traditional AST representations of the program structure, it finally converts the dependency graphs. This automated production phase (ii) needs to be executed for every project we analyze.

Phase (i) requires training data in form of correct dependency graphs for source code programs. Phase (ii) requires the conversion of dependency structures to AST representations. Therefore, we developed: (a) **Source Code ⇒ Syntax Tree**, an approach to generate syntax trees for correct and complete source codes of a programming language, (b) **Syntax Tree ⇒ Dependency Graph**, an approach to encode the syntax trees as dependency graphs, and (c) **Dependency Graph ⇒ Syntax Tree**, an approach to convert the dependency graphs back to syntax trees.

Step (a) uses traditional front-end technology and has
been accomplished according to [16]. Step (b) essentially captures the non-terminal names of the syntax tree in the edge labels of the dependency tree. Some additional information incorporated in the dependency tree is however necessary to perform step (c) without conversion errors. The general approach and software for steps (b) and (c) are described in details in [9], and have for this study been adapted from natural language parsing.

Moreover, the conversion to dependency structure in step (b) can be done in several different ways. Various strategies for forming the dependency structure are discussed in [16]. The results presented below simply use the strategy that turned out to produce the best accuracy for programming languages.

3. Experiments

This section presents the parsing experiments and evaluates the accuracy of the syntax trees produced by the parser. The parsing algorithm is robust but can commit errors even for correct input. This is investigated in the subsequent error analysis. We begin with the experimental setup.

Setup: The open-source software MaltParser is used in the experiments as the information extraction tool producing syntax trees (the dotted box in figure 1). It implements the dependency parser, as well as the conversions from syntax trees to dependency trees (b) and back (c). It comes with the machine learner LIBSVM, which requires training data. The source files of the following projects have been turn into training and testing data.

- For Java: Recoder 0.83 [8] were used (400 source files with 92k LOC and 335k tokens).
- For C/C++: Elsa 2005.08.22b [12], where all 1389 source files in the entire distribution were used (265k LOC and 691k tokens).
- For Python: Natural Language Toolkit 0.9.5 [3] (160 source files with 65k LOC and 280k tokens).

To construct the syntax tree for the source code files (a) as required in the training phase (i), we adopted different front-ends for the different programming languages. For Recoder, a Java source code parsing and manipulation tool, we could use Recoder itself. For Elsa, we adopted CDT 4.0.3, a C/C++ parser plug-in to the Eclipse IDE, for producing the abstract syntax trees. The Python 2.5 interpreter is shipped with an analyzer that produces concrete syntax trees which we have utilized for the python project above.

For the experiments, the source files are for each project divided into a training set and a production test set, where the former comprises 80% of the files and the latter 10%. The remaining 10% is left untouched for later use.

The standard evaluation metric for parse trees for natural languages based on context-free grammar is F-score, the harmonic mean of precision and recall. The evaluation metric compares subtrees derived from the test data with those derived from the parser. A subtree in the parser output matches a subtree in the test data when they span over the same terminals in the input string. Recall is the ratio of matched subtrees over all subtrees in the test data. Precision is the ratio of matched subtrees over all subtrees found by the parser. F-score comes in two versions, one unlabeled (F_U) and one labeled (F_L), where each correct subtree in the latter also must have the correct nonterminal name.

Accuracy Results: We will here present the parsing results. Table 1 shows the parsing accuracies of the conducted experiments for the concrete syntax trees.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Python</th>
<th>C/C++</th>
<th>≈C</th>
<th>Ger</th>
<th>Swe</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_U</td>
<td>99.5</td>
<td>98.3</td>
<td>96.5</td>
<td>94.9</td>
<td>81.4</td>
<td>76.8</td>
</tr>
<tr>
<td>F_L</td>
<td>97.8</td>
<td>98.2</td>
<td>96.1</td>
<td>94.2</td>
<td>78.7</td>
<td>74.0</td>
</tr>
</tbody>
</table>

Table 1. Parsing results.

The measured accuracy is close to 100%, with Java having the highest accuracy. We are not aware of any similar studies for programming languages. However, compared German and Swedish in [9] (also shown in table 1), the figures reported here for Java, Python and C/C++ are way better. It is however worth noting that natural languages are more complex and less regular compared to a programming language. On the other hand, some syntactic constructions can be difficult for the parser, as it may not be able to capture the necessary information in order to resolve them, such as pairwise matching brackets. We conjecture that these figures are sufficiently high for a large number of reverse engineering and program comprehension tasks.

As mentioned in the introduction, one benefit of this approach is that one rapidly can produce an analysis of source code of a C dialect using the above parser trained on another C dialect. This can be of importance for companies using proprietary C dialects who want to avoid the effort of also constructing specialized fact extractor for their reverse engineering and analysis tools. Hence, one additional experiment has been performed to get a rough estimation on how high accuracy we can expect for C project written in a C dialect, when the dependency parser has not been trained on source code for that dialect. Here the parser was trained using five different C/C++ projects (including Elsa) which in total generated 1590 concrete syntax trees having 859k tokens.

We evaluated the accuracy on a project with “real world” source code for a local company that uses such a compiler for a specialized C dialect. Their compiler uses among other things additional keywords. It is again worth noting that CDT failed to produce syntax tree for 9 of 85 source files (with 71k LOC) of the project, that is 4.5%. The test data in this case is the remaining 76 source files, having 153k
The figures are shown in the column ≈C of table 1, and are slightly lower than for the above presented result for C/C++, but still high enough to be useful. The drop in accuracy is expected partly because the various coding styles that different programmers use in the source code of training data compared to the source code of the test data.

**Performance Results:** The analysis time has not been prioritized in this study, and the time for the figures of the Java code in table 1 was about 2 source files per minute, with similar analysis time for the Python and C/C++ projects. However, there are several optimization techniques already implemented in MaltParser for reducing the time. Experiments on natural languages show that optimization pushes parsing speed often below 10%.

**Error analysis:** The result for Java for the dependency structure (before converting it to AST) will now be studied, in order to get a deeper insight into the types of errors that the parser causes. Specifically, the labeling mistakes caused by the parser are investigated here. This is done by producing a confusion matrix based on the dependency labels. That is, how often does a parser think that an arc has the label $X$ when it in fact should have been the label $Y$. This is shown in table 2 for the 3 most common errors. The table holds for the AST output as well, since the labels are conveyed to the nonterminal names [9].

<table>
<thead>
<tr>
<th>Freq.</th>
<th>Correct Label</th>
<th>Parsing Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>FieldReference</td>
<td>VariableReference</td>
</tr>
<tr>
<td>25</td>
<td>VariableReference</td>
<td>FieldReference</td>
</tr>
<tr>
<td>12</td>
<td>MethodDeclaration</td>
<td>LocalVariableDeclaration</td>
</tr>
</tbody>
</table>

**Table 2. Confusion matrix for Java.**

Looking at the two most frequent errors, we conclude that the parser confuses the labels *FieldReference* and *VariableReference*. A FieldReference refers to a class attribute whereas a VariableReference could refer to either an attribute or a local variable. The parser mixes a reference to a class attribute with a reference that could also be a local variable or vice versa. This is an error that type- and name-analysis can easily resolve. On the use-occurrence of a name (reference), analysis looks up for both possible define-occurrences of the name (declaration), first a *LocalVariableDeclaration* and then a *FieldDeclaration*, and uses the one that is found first.

Another type of confusion involves declarations, where a *MethodDeclaration* is misinterpreted as a *LocalVariableDeclaration*. This type of error can be resolved by a simple post-processing: a LocalVariableDeclaration followed by opening parenthesis (always recognized correctly) is a MethodDeclaration.

All these errors are by and large relatively harmless in the perspective of reverse engineering and program comprehension. One can in most cases easily look up the correct label in a post-processing step, in case it is of importance for the task at hand. Another solution can be to map all types of references to the same super label (e.g. Reference) during phase (i) and then split the label after the parse, if necessary. The same can be done for declarations.

It is also noteworthy that the parsing errors, corrected or not, are abstracted away in subsequent analyses as commonly used in program comprehension. For instance, without any further correction in a post-processing step, the two inheritance graphs of our test program, the correct one and the one constructed using the not quite correct parsing results, are identical. The same holds for the two type reference graphs.

### 4. Related Work in Reverse Engineering

Breadth-First Parsing [19] was designed to provide better error stabilization than traditional parser (generators) provide. It uses a two phase approach, where the first phase identifies high-level entities and the second phase parses the structure with these entities as root nonterminals (axioms). Fuzzy Parsing [11] was designed to efficiently develop parsers by performing the analysis on selected parts of the source instead of the whole input. It is specified by a set of (sub-)grammars, each with their own axioms. The actual approach is then similar to Breadth-First Parsing. It makes parsing more robust in the sense that it ignores source fragments. A prominent tool using the fuzzy parsing approach for information extraction in reverse-engineering tools is Sniff [4] for analyzing C++ code. Island grammars [14] generalize on Fuzzy Parsing. Parsing is controlled by two grammar levels (island and sea) where the sea-level is used when no island-level production applies.

Syntactic approximation based on lexical analysis was developed with the same motivation as our work: when maintenance tools need syntactical information but the documents could not be parsed for some reason, hierarchies of regular expression analyses could be used to approximate the information with high accuracy [15, 7]. A similarly robust and lightweight approach for information extraction constructs XML formats (JavaML and srcML) from C/C++/Java programs first, before further processing with XML tools like Xpath [2, 5]. It combines lexical and context-free analyses. Lexical pattern matching is also used in combination with context free parsing in order to extract facts from semi-structured specific comments and configuration specifications in frameworks [10].

TXL is a rule-based language defining information extraction and transformation rules for programs in formal languages [6]. It allows to incrementally extend the rule base and to adapt to language dialects and extensions. As the rules are context-sensitive, TXL goes beyond the lexical
and context-free approaches discussed before.

The fundamental difference of our approach compared to lexical, context-free, and -sensitive approaches (and combinations thereof) is that we use automated machine learning instead of manual specification for defining and adapting the information extraction.

5. Conclusions and Future Work

We applied a natural language parsing techniques to information extraction from formally structured information sources, such as programs. It offers robustness as it always produces some output even for incorrect input. Experiments even showed that, applied to Java, C/C++, and Python, the accuracy of parsing is close to 100%. Furthermore, the detailed error analysis showed that the errors are often simple mistakes which are forgiveable (since they are abstracted from in later processing phases of reverse-engineering) and easily correctable. The advantage of our approach over robust information extractors used so far is its rapid adaptability to new languages: instead of explicitly specifying the information extractor using (grammar and transformation) rules, we automatically generate the language specific information extractor using machine learning and training of a generic parsing approach. The training data can be generated automatically, as well. This could increase the development efficiency of parser variants since, no language specification, only examples are to provide.

Besides efficient information extractor development, efficient parsing itself is important in many applications. This is of less importance for natural language parsing since sentences are on average relatively short. Applied to programs which can easily contain several millions lines of code, a parser with more than linear time complexity is not acceptable. Our generic parser is linear (in contrast to many other natural language parsers) and processes example programs in acceptable time, as our experiments showed. In fact, the best results presented here beat the best parsing results for natural languages with a wide margin.

Although these results are promising, they are only the first step towards natural language parsing leveraging on information extraction for reverse-engineering. Our next step is to connect extracted results with client analyses, e.g., software metrics and architecture recovery. In fact, only in terms of these client analyses, we can ultimately evaluate the accuracy of our approach.

In practice, we want to apply our approach to more dialects of C/C++. We aim at experimentally evaluating the accuracy when analyzing correct, incomplete, and erroneous programs for both standard C and its dialects. The experiments with C/C++ presented in this paper are only a first step towards this goal. The application of natural language parsing for information extraction from formal language codes has the potential of a seamless integration with information extraction from natural language documents e.g., documentation and comments. This remains to be investigated in the future.

References