Växjö University

School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

Using MaltParser for Fact Extraction from Programs

Jens Nilsson, Welf Löwe, Johan Hall, Joakim Nivre

March
2009

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 09010
ISSN 1650-2647
ISRN VXU/MSI/DA/R/--09010/--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 [22]. 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. Robustness 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:

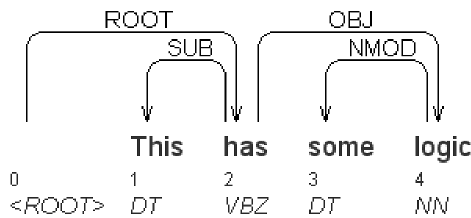

Figure 1: Sentence with a dependency graph.

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, especially, analyzing the accuracy of the extracted information.

In detail, section 2 discusses related work and gives an introduction to the information extraction tool. Section 3 describes the preparation of the training examples necessary for the NLP parser applied here, while section 4 presents the experiment results. Section 5 discusses related work in information extraction for reverse-engineering. We end with conclusions and future work in section 6.

2 NLP Background

The syntactic structure of formal languages, e.g., programming languages, is defined using context-free grammar containing both terminals and nonterminals. This is also the case for natural languages. **Dependency structure** is another way of representing the syntax of natural languages. Dependency trees form labeled, directed and rooted trees, as shown in the figure 1. One essential difference compared to context-free grammar is the absence of nonterminals. Another is that the syntactic structure is composed of lexical tokens (also called terminals or words) linked by binary and directed relations called *dependencies*. Each token in the figure is labeled with a word class, shown in the bottom of the figure. Each dependency relation is also labeled, *SUB* marks, e.g., the subject and *OBJ* the object.

The parsing algorithm, used in the experiments of section 4, can produce such dependency trees. It is in many ways similar to the shift-reduce parser for context-free grammars, with the most apparent difference that terminals (not nonterminals) are pushed onto the stack. Similar to the shift-reduce parser, the construction of syntactic structure is created by a sequence of transitions. It starts with an empty stack and terminates when the input queue is empty. The algorithm has a linear time complexity as it is guaranteed to terminate after at most $2n$ transitions, given that the length of the input sentence is n [26].

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 [27]. Hence, the parser requires benchmark data (also known as training data) containing 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 that can contain dependency trees for sentences of any language irrespectively of whether the language is a natural of formal one, 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 is bound to commit errors even if the input is acceptable according to a grammar. This is among other things a result of the fact that the amount of training data is finite.

3 General Approach

In section 2, we presented the parser for producing dependency graphs for natural languages. Here we will present how it can be used for producing syntactic structure for programming languages. Since the framework requires training data in form of correct dependency graphs, we need an approach for converting source code to dependency graphs. The evaluation of the syntactic structure produced by the parser is presented in section 4.

Our general approach can be divided into two phases, training and production. In order to be able to perform both these phases in this study, we need to adapt natural language parsing to the needs of information extraction from programming language codes, i.e., we needed to automatically produce training data. 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 adaptation of an existing approach for encoding the syntax trees as dependency graphs to programming languages.

(c) **Dependency Graph ⇒ Syntax Tree**: an adaptation of an existing approach to convert the dependency graphs back to syntax trees.

These approaches have been accomplished as presented below. In phase (i), we need to train and adapt the generic parsing approach to a specific programming language. Therefore:

(1) **Generate training data** automatically by producing syntax trees and then dependency graphs for correct programs using approaches (a) and (b).

(2) **Train** the generic parser with the training data.

This automated deployment phase (i) needs to be done for every new programming language we adapt to. We have done it for Java, Python and C/C++. Finally in phase (ii), we extract the information from (not necessarily correct and complete) programs:

(3) **Parse** the new source code into dependency graphs.

(4) **Convert** the dependency graphs into syntax trees using approach (c).

This automated production phase (ii) needs to be executed for every project we analyze.

Phase (i) has already been discussed in section 2 for parsing natural languages, and can be generalized to parsing programming languages, once the approaches (a)–(c) has been accomplished. Therefore, we present steps (a)–(c) in the sections 3.1, 3.2, and 3.3, respectively. Experimental results of phase (iii) are discussed in section 4.

3.1 Source Code ⇒ Syntax Trees

The parsing algorithm described in section 2 has been developed for parsing natural languages, which makes it necessary to resolve a number of issues that arise when the parser instead is adapted for source code as input. One relatively unproblematic issue is how we define a word in a programming language, where we will simply let a word in a natural language be equivalent to a token in a programming language. One slightly more problematic issue is how to define a "sentence" in source code.

A natural language text syntactically decomposes into a sequence of sentences in a relatively natural way. But is there also a natural way of splitting source code into sentences? The most apparent approach may be to define a sentence as a compilation unit, that is, a file of source code. This can however result in practical problems since a sentence in a natural language text is usually on average between 15–25 words long, partially depending on the author and the type of text. The sequence of tokens in a source file may on the other hand be much longer. Time complexity is usually in practice of less importance when the average sentence length is as low as in natural languages, but that is hardly the case as when there can be several thousands tokens in a sentence to parse.

Other approaches could for instance be to let one method be a sentence. However, then we need to deal with other types of source code constructions explicitly. We have in this study for simplicity let one compilation unit be one sentence. This is possible in practice due to the linear time complexity of the parsing algorithm of section 2, a quite unusual property compared to other NLP parsers guided by machine learning with state-of-the-art accuracy.

In order to produce training data for the parser for a programming language, an analyzer that constructs syntax trees for correct and complete source code of the programming language is needed. We are in this study focusing on Java, Python and C/C++ and consequently need one such an analyzer for each language. For example, figures 2 and 3 show the concrete syntax tree of the following snippets of Java:

```
  Example (1):
public String getName() {
   return name;
}

  Example (2):
while (count > 0) {
   stack[--count]=null;
}
```

All source code comments and indentation information (except for Python where the indentation convey hierarchical information) have been excluded from the syntax trees. All string and character literals have also been mapped to "string" and "char", respectively. This does not entail that the approach is lossy, since all this information can be retained in a post-processing step, if necessary. As pointed out by for instance [6], comments

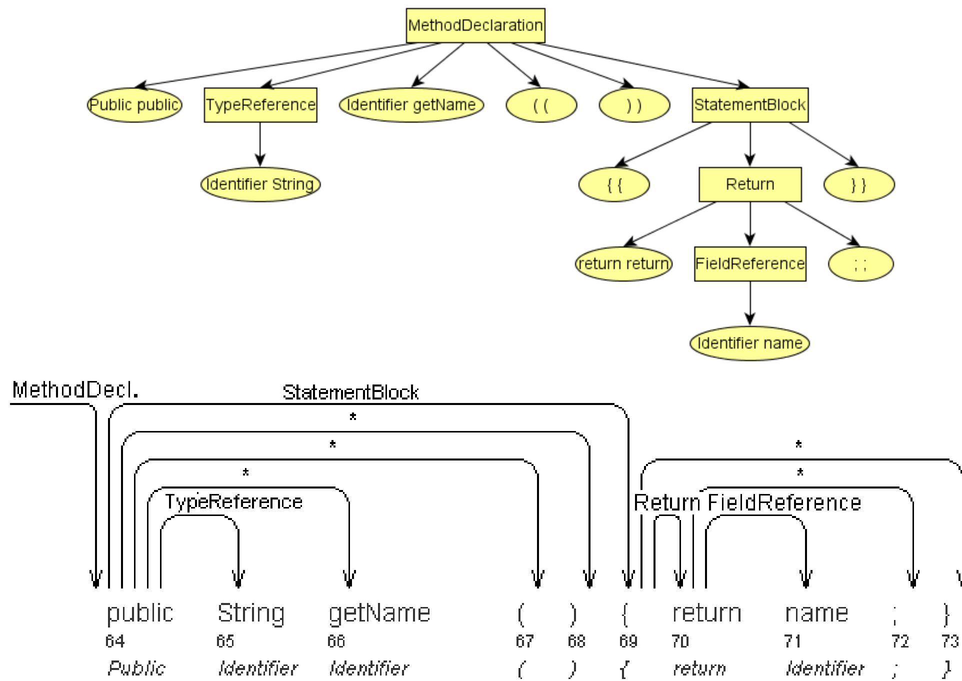

Figure 2: Syntax tree and dependency graph for Example (1).

and indentation may among other things be of interest when trying to understand source code in reverse engineering.

3.2 Syntax Trees ⇒ Dependency Graphs

Here we will discuss the conversion of syntax trees into dependency graphs. Any context-free grammar can be converted into a dependency grammar [15]. This means that any syntactic tree based on context-free grammar also can be converted into a well-formed dependency graph.

We use a method that has been successfully applied for natural languages for converting syntax trees into a *reconvertible* dependency graph that makes it possible to perform the inverse conversion, where "reconvertible" means that information about the syntax tree is saved in complex arc labels [13]. We also present results in section 4 where the dependency graph cannot be used for the inverse conversion, which we call a *non-reconvertible* dependency graph.

The conversion is performed in a two-step approach. First, the algorithm traverses the syntax tree from the root and identifies the head-child and the terminal head for all nonterminals in a recursive depth-first search. To identify the head-child for each nonterminal, the algorithm uses heuristics called *head-finding rules*. Two head-finding strategies have been investigated. For each nonterminal:

1. Let the leftmost terminal child be the head of all other element in the nonterminal. If no terminal child can be found, the head-child of the nonterminal will be the leftmost nonterminal child and the terminal head will be the terminal child of the head-child recursively.

2. Let the leftmost terminal in the entire subtree of the nonterminal be the head of all other elements.

The dependency graphs in figures 2 and 3 use the second head-finding strategy, which for instance induces that all arcs are pointing to the right. Second, a dependency graph is created according to the identified terminal heads. The arcs in the reconvertible dependency graph are labeled with complex arc labels, where each complex arc label consists of two sublabels:

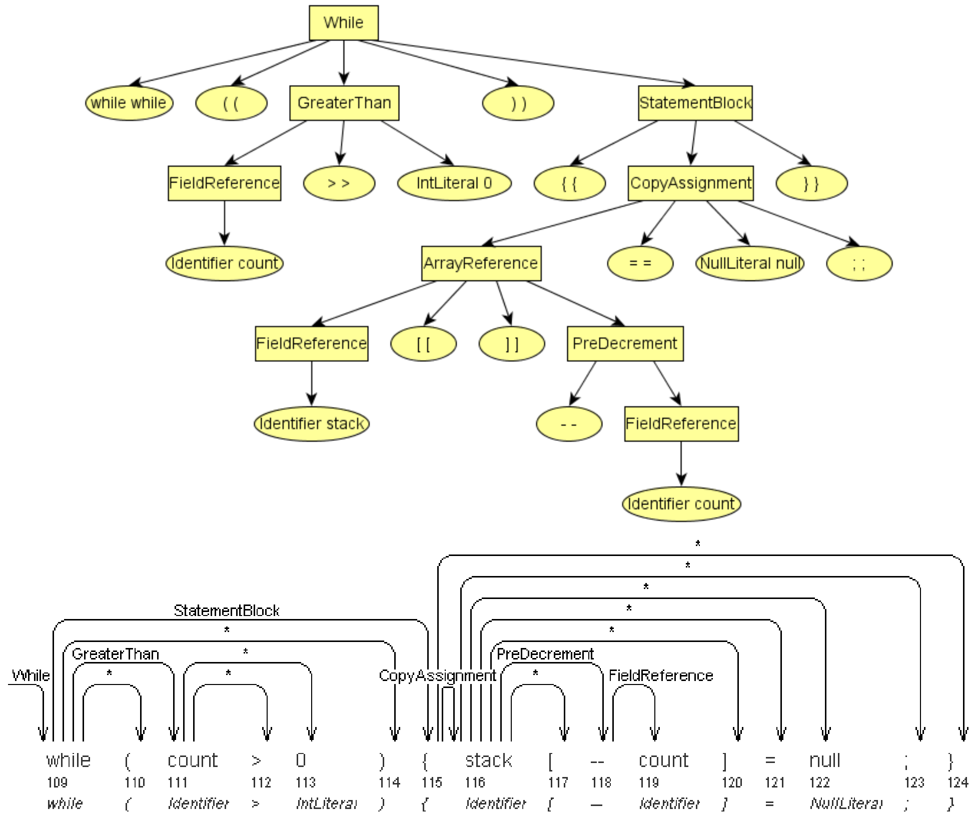

Figure 3: Syntax tree and dependency graph for Example (2).

1. Encode the dependent spine, i.e., the sequence of nonterminal labels from the dependent terminal to the highest nonterminal where the dependent terminal is the terminal head (the nonterminal labels are separated by |),
2. Encode the attachment point in the head spine, a non-negative integer value a, which means that the dependent spine is attached a steps up in the head spine.

By encoding the arc labels with these two sublabels, it is possible to perform the inverse conversion and how this done is explained in subsection 3.3.

The non-reconvertible dependency labels allow us to reduce the complexity of the arc labels, making the learning problem simpler due to fewer distinct arc labels. This may result in a higher accuracy during parsing and can be used as input for further processing directly without taking the detour back to syntax trees. This can be motivated by the fact that all information in the syntax trees is usually not needed anyway in many reverse engineering tasks. Each dependency arc is the highest nonterminal name of the spine, that is, the single nonterminal name that is closest to its head. The non-reconvertible dependency label also excludes the attachment point value, making the learning problem even simpler. Figures 2 and 3 show the non-reconvertible dependency labels of the syntax trees in the same figures, where each label contains just a single nonterminal name of the original syntax trees.

3.3 Dependency Graphs ⇒ Syntax Trees

The inverse conversion is a bottom-up and top-down process on the *reconvertible* dependency graph (must contain complex arc labels). First, the algorithm visits every terminal in the reconvertible dependency graph and restores the spines of nonterminals with labels for

each terminal using the information in the first sublabel of the incoming arc. Thus, the bottom-up process results in a spine of zero or more arcs from each terminal to the highest nonterminal of which the terminal is the terminal head. Secondly, the spines are weaved together according to the arcs of the dependency graph. This is achieved by traversing the dependency graph recursively from the root using a pre-order depth-first search, where the dependent spine is attached to its head spine or to the root of the syntax tree. The attachment point a, given by the second sublabel, specifies the number of nonterminals between the terminal head and the attachment nonterminal.

4 Experiments

We will in this section present the parsing experiments and evaluate the accuracy of the syntax trees produced by the parser. As mentioned in section 2, the parsing algorithm is robust in the sense that it always produces a syntactic analysis no matter the input, but it can commit errors even for correct input. This is investigated in the subsequent error analysis. But we begin with the experimental setup.

4.1 Experimental Setup

The open-source software MaltParser (maltparser.org) will be used in the experiments as the information extraction tool producing syntax trees. It contains an implementation of the parsing algorithm, as well as an implementation of the conversion strategy from syntax trees to dependency trees and back, presented in subsections 3.2 and 3.3. It comes with the machine learner LIBSVM [5], producing the most accurate results for natural language parsing compared to other evaluated machine learners [14]. LIBSVM requires training data, where source files of the following projects have been converted into dependency graphs.

- For Java: Recoder 0.83 [11] were used, using all source files in the directory "src" (having 400 source files with 92k LOC and 335k tokens).
- For C/C++: Elsa 2005.08.22b [21], where 1389 source files were used, including the 978 C/C++ benchmark files shipped in the distribution (thus comprising 1389 source files with 265k LOC and 691k tokens).
- For Python: Natural Language Toolkit 0.9.5 [3], where all source files in the directory "nltk" were used (having 160 source files with 65k LOC and 280k tokens).

To construct the syntax tree for the source code file of Recoder, we have used Recoder. It creates an AST for source file, but we are currently interested in the concrete syntax tree with all the original tokens. In this first conversion step, the tokens of the syntax trees are thus retained. For example, the syntax trees in figures 2 and 3 are generated by Recoder.

The same strategy was adopted for Elsa with the difference that CDT 4.0.3, a plug-in to the Eclipse IDE to produce syntax trees for source code of C/C++, was used for producing the abstract syntax trees.[1] It produces abstract syntax trees just like Recoder, so the concrete syntax trees have also been created by retaining the tokens.

The Python 2.5 interpreter is actually shipped with an analyzer that produces concrete syntax trees (using the python imports `from _ast import PyCF_ONLY_AST` and `import parser`), which we have utilized for the python project above. Hence, no additional processing is needed in order prepare the concrete syntax trees as training data.

For the experiments, the source files have been divided into a training set T and a development test set D, where the former comprises 80% of the dependency graphs and the latter 10%. The remaining 10% (E) has been left untouched for later use. The source files have been ordered alphabetically by the file names including the path. The dependency graphs have then been distributed into the data sets in a pseudo-randomized way. Every tenth dependency graph starting at index 9 (i.e. dependency graphs 9, 19, 29, ...) will belong to D, and every tenth dependency graphs starting at index 0 to E, while the remaining graphs constitute the training set T.

4.2 Metrics

The standard evaluation metric measuring accuracy for dependency parsing for natural language is labeled (AS_L) and unlabeled (AS_U) attachment score. AS_U measures how correct the dependency structure is, and

[1] It is worth noting that CDT failed to produce syntax trees for 2.2% of these source files, which were consequently excluded from the experiments. This again indicates the difficult of parsing C/C++ due to its different dialects.

is the ratio of tokens attached to its correct head. AS_L is the same as AS_U with the additional requirement that the dependency label should be correct as well.

The standard evaluation metrics for parse trees for natural languages based on context-free grammar is F-score, the harmonic mean of precision and recall. The evaluation metrics compare 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.

4.3 Results

We will here present the parsing results. Table 1 shows the parsing accuracies for the conducted experiments, with one language per row. The RECON columns show AS_U and AS_L with the reconvertible dependency labels. The ST columns show the F-score for the concrete syntax trees, whereas the NON-RECON columns present the figures using non-reconvertible dependency labels. As mentioned in section 3.2, a number of different head-finding strategies have been evaluated, and the presented figures use the best ones.

We are not aware of any similar studies for programming languages so we start by comparing the results to natural language parsing. First of all, the accuracies for dependency structure in the RECON columns are better than figures reported for natural languages. Some natural languages are easier to parse than others, and the parsing experiments in [12] for dependency structure indicate that English and Chinese are relatively easy, with AS_L close to 89% and AS_U around 87%. We see that the highest accuracies are recorded for Java and the lowest for C/C++. For a large number of reverse engineering and program comprehension tasks, all results are likely to be sufficiently high. This statement is strengthened by the errors analysis in subsection 4.4, indicating that many of the errors are harmless.

Turning our attention to the ST columns, the relationship between the programming languages is the same as for RECON, with Java having the highest accuracy. Again, we are not aware of any similar studies for programming languages, but compared to parsing German, with F_U = 81.4% and F_L = 78.7%, and Swedish, having F_U = 76.8 and F_L = 74.0 [13], the figures reported here 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 a data-driven parser in case the feature model in not able to capture the necessary information in order to resolve them, such as pairwise matching brackets. We conjecture again that these figures are sufficiently high for a large number of reverse engineering and program comprehension tasks.

As mentioned in section 3, the advantage of producing concrete syntax trees is that well-know techniques in reverse engineering and program comprehension can be applied in order to perform program analysis. However, since a lot of information often is abstracted away from the concrete syntax trees in these tasks. It is therefore of interest if the amount of information that the parser produces can be reduced in order to simplify the learning problem, which could increase the accuracy for the dependency structure even though a conversion back to concrete syntax trees is no longer possible. It is therefore noteworthy that the accuracy for Java using NON-RECON increases up to 99.7% for AS_U and above 99% for AS_L. These results indicate that it can be of interest to use this output instead of the concrete syntax trees for Java. The accuracy for does not increase for Python and C/C++ using NON-RECON compared to ST, entailing the additional information is actually of importance for the feature model, despite a more complex the learning problem.

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

4.4 Error Analysis

This subsection will study the result for Java with NON-RECON, 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

	ReCon		St		Non-ReCon	
	AS_U	AS_L	F_U	F_L	AS_U	AS_L
Java	98.7	97.6	99.5	97.8	99.7	99.1
Python	96.5	95.8	98.3	98.2	96.3	95.6
C/C++	94.3	93.6	96.5	96.1	93.7	93.2

Table 1: Parsing Results.

Freq.	**Correct Label**	**Parsing Label**
66	FieldReference	VariableReference
25	VariableReference	FieldReference
12	MethodDeclaration	LocalVariableDeclaration
9	Conditional	FieldReference
5	NotEquals	MethodReference
4	Plus	MethodReference
4	Positive	*
4	LessThan	FieldReference
4	GreaterOrEquals	FieldReference
4	Divide	FieldReference
4	Modulo	FieldReference
4	LessOrEquals	FieldReference

Table 2: Confusion matrix for Java with Non-ReCon.

the 15 most common errors, ordered by descending frequency of error types.

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*.

Errors that involve binary operators, e.g., *Conditional*, *NotEqual*, *Plus*, are at rank 4 and below in the list of the most frequent errors. They are likely a result of the incremental left-to-right parsing algorithm. The whole expression should be labeled with as a binary operator whereas it is labeled as a *MethodReference* or *FieldReference* instead. The references actually occur in the left-hand side sub-expression of the binary operators. This means that subexpressions and bracketing were recognized correctly but, the type of the top expression node was mixed up. Extending the lookahead of the input queue, making it possible for the classifier in the parser to look at even more yet unparsed tokens, might be one possible solution. However, these errors are by and large relatively harmless anyway. Hence, no correction is taken.

Figure 4 displays some typical mistakes for the input

```
return (fw.unitIndex == unitIndex &&
        fw.unitIndex.equals(unitList));
```

The parser mixes up a *ParenthesizedExpression* with a *Conditional*, a boolean *ParenthesizedExpression* only occurring in conditional statements and expressions, which is forgivable. Then it incorrectly assigns the label *Equals* to the arc between the first left parenthesis and the first `fw` instead of the correct label *LogicalAnd*. It mixes up the type of the whole expression, an *Equals*-is taken for an *LogicalAnd*-expression, which is forgivable, as well. Finally, the two *FieldReference*s are taken a more general *VariableReference*s, which is correctable as discussed.

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, the correct one and the one constructed using the not quite correct parsing results, are identical.

4.5 Parsing Source of a C Dialect

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

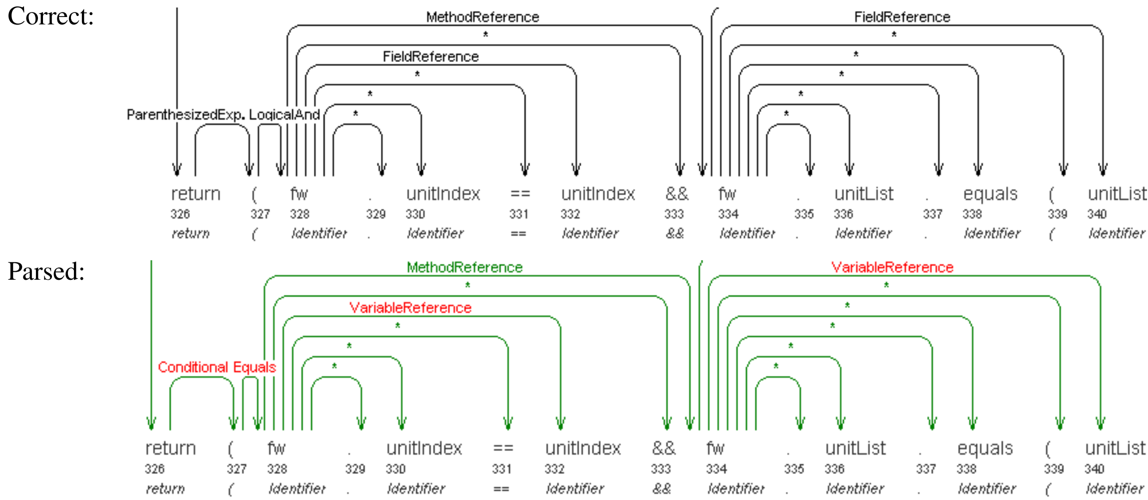

Figure 4: Typical errors by NON-RECON.

trained on another C dialect. This can be of importance for companies which does not have the time and effort to construct a specialized fact extractor for their specific compiler. 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.

As a use case, we have 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%, which partly is due to problems of CDT coping with this C dialect. The test data in this case is the remaining 76 source files, having 153k tokens in total.

The accuracy for this experiment is F_U=94.9 and F_L=94.2, which is 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. We find these results useful as they are, as the accuracy for the remaining 4.5% of the code is likely more or less as high. However, since we are unable to produce the correct syntax trees using CDT, we cannot present their accuracies.

5 Related Work in Reverse Engineering

Front-ends for formal languages have been known for many years. It is an established practice to implement them in a pipelined architecture with scanner, parser and semantical analysis components accepting a regular superset of the language, a context-free superset, and the actual language, respectively (cf. text books on compiler construction, e.g. [31]). While this established architecture and design technology simplifies front-end (re-)engineering, it is still a difficult and time consuming task.

All three components and complete front-ends can be generated out of appropriate specifications: regular expressions (scanner), context free grammars (parser), and attributed grammars (semantic analysis) including the generation of error stabilization for parsers [16]. Generator technology simplifies front-end development even more. Still front-end (re-)engineering requires the understanding of complex specification languages and defining/modifying complex specifications. Moreover, error stabilization often throws away large parts of the source – it is robust but does not care about maximizing accuracy.

Breadth-First Parsing [28] was designed to provide

better error stabilization then traditional parser (generators) provide. It uses a two phase approach: the first phase identifies high-level entities – the second phase parses the structure with these entities as root nonterminals (axioms).

Fuzzy Parsing [18] 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 scans for instances of the axioms and then parses according to the grammar. It makes parsing more robust in the sense that it ignores source fragments – including missing parts, errors and deviations therein – that subsequent analyses abstract from anyway. A prominent tool using the fuzzy parsing approach for information extraction in reverse-engineering tools is Sniff [4] for analyzing C++ code.

Island grammars [23] 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. The island-level corresponds to the sub-grammars of fuzzy parsing. Island grammars have been applied in reverse-engineering, specifically, to bank software [24].

Syntactic approximation based 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 [25, 8]. Their information extraction approach is characterized as "lightweight" in the sense that it requires little specification effort.

A similar robust and light-weight 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, 6]. 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 [17].

There is documentary information in source code in addition to its formal structure according to a programming language [30], e.g., contained comments and indentation. Information extraction for reverse engineering and program comprehension ought not to loose this information. It is therefore deliberately captured in formats like JavaML and srcML. Our approach is not lossy in this respect either, since all this information can be retained in a post-processing step, if necessary.

TXL is a rule-based language defining information extraction and transformation rules for programs in formal languages [7]. 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.

General NLP techniques have been applied for extracting facts from general source code comments to support the understanding of programs [9]. Comments are extracted from source code using classic lexical analysis; additional information is extracted (and then added) with classic compiler front-end technology.

NLP has also been applied to other information extraction problems in reverse-engineering: to analyze unstructured, large information sources. For instance, it is used to reverse-engineer requirement specifications [29], in clone detection [19], and to connect program documentation to source codes [20].

While structured reverse-engineering techniques have been applied to unstructured information sources, e.g., to program documentation [10], to the best of our knowledge, our work is the first applying natural language parsing to information extraction from formally structured information sources. 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.

6 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, at the price of a small amount of errors even for correct 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 left are often simple mistakes which are forgivable (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 easy 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

[1] Paul Anderson. 90 % perspiration: Engineering static analysis techniques for industrial applications. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 3–12, 2008.

[2] Greg J. Badros. JavaML: a markup language for Java source code. In *Proceedings of the 9th International World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, pages 159–177. North-Holland Publishing Co., 2000.

[3] Steven Bird, Edward Loper, and Ewan Klein. Natural Language Toolkit (NLTK) 0.9.5. http://nltk.org/, 2008.

[4] Walter R. Bischofberger. Sniff: A Pragmatic Approach to a C++ Programming Environment. In *C++ Conference*, 1992.

[5] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines, 2001.

[6] Michael L. Collard, Huzefa H. Kagdi, and Jonathan I. Maletic. An XML-Based lightweight C++ fact extractor. In *11th IEEE International Workshop on Program Comprehension*, pages 134–143. IEEE Computer Society, 2003.

[7] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: a rapid prototyping system for programming language dialects. *Comput. Lang.*, 16(1):97–107, 1991.

[8] Anthony Cox and Charles L. A. Clarke. Syntactic Approximation Using Iterative Lexical Analysis. In *Proceedings of the 11th International Workshop on Program Comprehension*, pages 154–163, 2003.

[9] Letha H. Etzkorn, Lisa L. Bowen, and Carl G. Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(3):219–236, 1999.

[10] G. Ewart and M. Tomic. Experiences using reverse engineering techniques to analyze documentation. In *3rd International Workshop on Program Comprehension*, 1994.

[11] Tobias Gutzmann, Dirk Heuzeroth, and Mircea Trifu. Recoder 0.83. http://recoder.sourceforge.net/, 2007.

[12] Johan Hall, Jens Nilsson, Joakim Nivre, Gülşen Eryiğit, Beáta Megyesi, Mattias Nilsson, and Markus Saers. Single Malt or Blended? A

Study in Multilingual Parser Optimization. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL*, pages 933–939, 2007.

[13] Johan Hall and Joakim Nivre. Parsing Discontinuous Phrase Structure with Grammatical Functions. In *Proceedings of 6th International Conference on Natural Language Processing*, pages 169–180. Springer, 2008.

[14] Johan Hall, Joakim Nivre, and Jens Nilsson. Discriminative classifiers for deterministic dependency parsing. In *21st International Conference on Computational Linguistics*, pages 316–323, 2006.

[15] D. G. Hays. Dependency theory: A formalism and some observations. *Language*, 40:511–525, 1964.

[16] Uwe Kastens, Anthony M. Sloane, and William M. Waite. *Generating Software from Specifications*. Jones and Bartlett Publ, 2007.

[17] Jens Knodel and Martin Pinzger. Improving fact extraction of framework-based software systems. In *10th Working Conference on Reverse Engineering*, pages 186–195. IEEE Computer Society, 2003.

[18] Rainer Koppler. A systematic approach to fuzzy parsing. *Softw. Pract. Exper.*, 27(6):637–649, 1997.

[19] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 107. IEEE Computer Society, 2001.

[20] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135. IEEE Computer Society, 2003.

[21] Scott McPeak. Elsa: The elkhound-based C/C++ parser. http://www.cs.berkeley.edu/∼smcpeak, 2005.

[22] Audris Mockus. Software Changes and Software Engineering: Why Not? Dagstuhl Seminar 05261: Multi-Version Program Analysis, Jun/Jul 2005.

[23] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, page 13. IEEE Computer Society, 2001.

[24] Leon Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension*. IEEE Computer Society Press, June 2002.

[25] Gail C. Murphy and David Notkin. Lightweight source model extraction. *SIGSOFT Software Engineering Notes*, 20(4):116–127, 1995.

[26] Joakim Nivre. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies*, pages 149–160, 2003.

[27] Joakim Nivre, Johan Hall, and Jens Nilsson. Memory-based dependency parsing. In H. T. Ng and E. Riloff, editors, *Proceedings of the 8th Conference on Computational Natural Language Learning*, pages 49–56, 2004.

[28] John Ophel. Breadth-First Parsing. citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.3035, 1997.

[29] Pete Sawyer, Paul Rayson, and Roger Garside. REVERE: Support for Requirements Synthesis from Documents. *Information Systems Frontiers*, 4:343–353(11), September 2002.

[30] Michael L. Van De Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, 2002.

[31] William Waite and Gerhard Goos. *Compiler Construction*. Springer, Jan 1984.

Växjö
universitet

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

tel 0470-70 80 00, fax 0470-840 04
www.msi.vxu.se