Detection of Intentions in Context-Free Grammars

Torsten Görg

Institute of Software Technology
University of Stuttgart
Universitaetsstr. 38, 70569 Stuttgart, Germany
torsten.goerg@informatik.uni-stuttgart.de

Abstract

A programming language is based on several intentions and design requirements. As this information is reflected in the grammar of the language some intentions can be reverse engineered from the grammar. This paper presents an approach to detect intention instances in context-free grammars to support understanding of the design of languages. Some typical intentions in imperative languages are shown, followed by a discussion how to detect them automatically with a static analysis tool based on graph transformation.

1 Introduction

In software product lines the development process does not only encompass the realization of systems but also the development of domain-specific languages (DSL) as production assets. As a consequence, reengineering is extended to language-related topics.

For reegineering such languages, it is necessary to detect the intentions that are expressed in the language design. Usually the syntax of programming languages is specified with context-free grammars. The language for defining grammars can be described in its elementary form by the metamodel shown in Figure 1 as a UML class diagram.

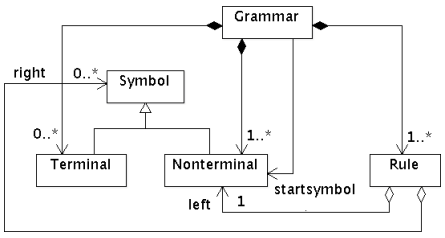

Figure 1: Metamodel for context-free grammars

The intentional programming paradigm has introduced the term intention to denote language abstractions [3]. Usually related to programming languages, here this approach is applied to BNF. The metaclasses in Figure 1 are the basic intentions in BNF. Similar to intentional programming, a set of new higher-level intentions has been defined to represent language design abstractions and capture common patterns and idioms. A higher-level intention consists of intentions at lower levels. This forms chains of intentions. At the highest level are the external requirements. Each detail in a grammar can ultimately be mapped to an external requirement. Such mappings are discussed for system implementations in [5]. Recognizing instances of higher-level intentions and their chaining helps to understand a given grammar.

2 Intentions in Grammars

A nonterminal on the right-hand side of a grammar rule can express different intentions, i.e., recursions which are further differentiated into recursions at the left or right end of a rule and mid recursions. Recursions are either direct or indirect, spanning the rules of several nonterminals. End recursions are used to realize sequences of similar repetitions, e.g., statement sequences. End recursions can also express sequences of elements separated by a delimiter, e.g., comma-separated sequences of parameter values. In a further intention level, sequences can express consecutive binary operations, e.g., the addition of summands. A typical pattern to specify operator precedences is to cascade several grammar rules for separate sequences. Figure 2 gives an example from the C++ grammar.

```
logical_and_expression ::= inclusive_or_expression |
    logical_and_expression "&&" inclusive_or_expression .

logical_or_expression ::= logical_and_expression |
    logical_or_expression "||" logical_and_expression .
```

Figure 2: Operator precedence in the C++ grammar

Mid recursions can be used for hierarchical compositions, following the composite design pattern [4]. Typical examples are parenthesized subexpressions and compound statements.

Other nonterminals stand for identifiers. Generally the intention in using identifiers is to establish associations between objects. Some of the identifiers appearing in grammar rules are related to declarations and give objects a name. Other occurences of identifiers are references to the named objects.

Some rules or parts of rules express enclosures. Typical examples are parameters surrounded by parenthesis and array indices in brackets. Further in-

tentions refer to the specialization of categories and to the specification of role names.

3 Detection of Intentions

The detection of an intention instance can be viewed as a graph transformation step. At first a pattern matching is required in the graph representing the grammar to find a part that is typical for a particular intention. After that the context of a candidate subgraph has to be checked. If the context fulfills all preconditions necessary for the intention the subgraph is replaced by another one that expresses the detected intention explicitly. Subsequent transformation steps are performed on the resulting graph incrementally for each recognized intention instance.

Intentions at higher levels are detected using patterns that contain underlying intentions recognized in preceded transformation steps. To simplify the pattern matching the detection is processed bottom-up, beginning with the elementary intentions.

```
PRECEDENCE_CHAIN(
    logical_and_expression ::=
        OPERATION( logical_and_expression, "&&", -> ) .
    logical_or_expression ::=
        OPERATION( logical_and_expression, "||", -> ) . )
```

Figure 3: Recognized precedence chain

The intention recognized first in the grammar fragment in Figure 2 is direct left recursion. Based on the recursions, sequences can be detected that are separated by the operators `&&` and `||`, respectively. The pattern for separated sequences is a rule with two alternatives that are identical, except for an additional separator symbol and an end recursion in one of the alternatives. The separated sequences in the example are further specialized to binary operation constructs. The cascading of the operators indicates a precedence chain. Figure 3 shows the result of the transformations. The meaning of the arrows is that the operations are evaluated from left to right.

4 Prototype for Intention Detection

I have implemented a prototype that analyzes context-free grammars statically. It takes a text file containing an arbitrary EBNF grammar as input. A front end parses the grammar and provides an object graph that is analyzed by a back end. A graph transformation is performed for each intention instance that is detected.

The prototype is realized as an Eclipse project, using Xtext [1] to generate a parser based on an EBNF metagrammar that conforms to the metamodel in Figure 1. Concepts for the higher-level intentions are added to extend EBNF to a language that can express input grammars as well as grammars that are enriched with intention information (Figure 3 is an example for the extended EBNF). The EBNF extention facilitates incremental graph transformations. Algorithms for pattern matching, context checking, and replacing subgraphs are implemented in plain Java code. Detection functions for different kinds of intentions are processed successively.

Currently the prototype can recognize all intentions mentioned above except the identifier related intentions and hierarchical compositions. The graph transformation approach makes it possible to add intention information manually as well. Automatic and manual analyses can be combined and provide an integral result. Information that is added manually in advance is used during an automatic analysis. Furthermore this has the advantage that the additions will not be lost for subsequent analyses.

5 Analysis Results

The prototype has analyzed the grammars of the common programming languages ANSI C++ and Java 1.6 [2]. All recursions and enclosures in these grammars have been recognized. In the C++ grammar 16 of 31 sequences, 16 of 23 separated sequences, 16 of 23 binary operations, 28% of the operator precedence chain, 19 of 27 specializations, and 13 of 34 role names have been detected. In the Java grammar 29 of 38 sequences, 21 of 29 separated sequences, 21 of 29 binary operations, 31% of the operator precedence chain, 27 of 42 specializations, and 9 of 39 role names have been detected. No false positives have occured.

6 Conclusion

It has been shown that several intentions can be detected automatically from an EBNF grammar. The proposed approach integrates definitions from an input grammar with results from automatic and manual analyses by extending the EBNF grammar language.

More intention instances could be detected if the pattern matching would recognize not only exact but also similar matches. And a grammar could be divided into rule subsets, based on the rule dependencies and recursion circles, to detect grammar components. The goal is to recognize different kinds of components, e.g., for expressions, for declarations, or for statements. The kinds of components in a grammar could indicate the paradigm of the analyzed language.

References

[1] *Xtext, v1.0.2.* http://www.eclipse.org/Xtext/, 2011.

[2] R. Bosworth. *The Grammar of the Java Programming Language, version 1.6, corrected.* http://www.it.bton.ac.uk/staff/rnb/bosware/javaSyntax/, 2011.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley Longman, 2000.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissidis. *Desin Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, 1994.

[5] M. Trifu. *Tool-Supported Identification of Functional Concerns in Object-Oriented Code.* PhD thesis, Karlsruher Institut fuer Technologie (KIT), Jan. 2010.