Model-aware Language Specification with Java

Jaroslav Porubän, Sergej Chodarev

Abstract—**Tools that support development of parsers often concentrate on concrete syntax, leaving abstract syntax defined only implicitly. On the other hand there are projectional language workbenches that give central role to language model (abstract syntax) at the cost of locking a language to the concrete tool. In this paper we present YAJCo parser generator that uses model-centered approach to language definition while preserving textual representation. Abstract syntax of a language is expressed using object-oriented model in a general-purpose language with additional information and concrete syntax provided in form of annotations. In the paper we describe how abstract syntax, concrete syntax and semantics of the language are defined using YAJCo. We also describe how this method supports language composition and iterative development.**

Index Terms—**language patterns, abstract syntax, domain-specific languages, parser generators, YAJCo**

I. Introduction

Development of computer languages and parsers is a well-researched topic. A lot of mature and established techniques and tools exist that facilitate this problem. However, development of a parser is still perceived as a hard problem and many domain-specific languages are developed based on some existing language. This led us to the development of a tool that makes development of languages more convenient for programmers with knowledge of object-oriented methodology. This means that it must be less concerned on grammars and parsing algorithms, but instead centered around definition of a language domain model.

In most situations while developing language processor, some kind of object-oriented domain model of the language is defined. During the parsing, instance of this model is created in memory based on input sentence as a collection of interconnected objects. This model (called *semantic model* by Fowler [1]) encapsulates all information needed to further process the sentence.

In most cases, structure of the model closely corresponds to language syntax. We can consider the model itself to represent abstract syntax of the language as it contains definition of language concepts and relations between them. This means that the model itself already contains a lot of information about syntax and can be used as a basis for language definition.

In our approach, the language model is expressed using a general-purpose language (GPL). We chosen Java as a widely used object-oriented language, although the approach is more generic. GPL like Java already provides all means for expressing object-models and also their semantics. In addition, it is well understood by programmers and supported by numerous tools lowering the learning costs. The tools allow to solve many common tasks, like refactoring, without the need to develop a special-purpose tooling (for example [2]).

The fact that language model is defined using "plain old Java objects (POJO)" also means that it is independent from parsing technology. This is in contrast with approaches based on projectional editing, like JetBrains MPS [3], where definition of language model and all its aspects is locked in the corresponding tool.

Object-model, however, does not contain definition of concrete syntax. This information can be added using annotations, that provide general-purpose mechanism for adding metadata to elements of the program. Therefore, annotations allow to integrate formal description with GPL, making it suitable for automatic analysis in a similar way as BNF or Petri nets [4].

We use this approach in a tool called YAJCo[1] (Yet Another Java Compiler Compiler) that allows to generate language parser based on annotated model [5]. Our goal was not to introduce new parsing methods. Instead, we integrated existing methods into a tool that allows language developer to work on higher level of abstraction with language definition that is independent on parsing algorithm and based on abstract syntax.

II. YAJCo Overview

As is obvious from the name of our tool, it relies on Java as a language for definition of language model and its semantics. Language model consists of annotated Java classes and relations between them. The model corresponds to abstract syntax of the language where each class represents a language concept.

The architecture of YAJCo is depicted in Fig. 1. The tool contains Java annotation processor that collects annotations attached to classes and their elements. It constructs an internal model of language definition based on that and uses it to generate a parser. YAJCo is able to extract relations between classes of the language model and infer part of language syntax based on that. Missing information about concrete syntax is derived from annotations.

This definition is used by YAJCo to generate parser specification for one of the existing parser generators. Currently JavaCC (top-down) and Beaver (bottom-up) are supported as backends making it possible to choose parsing algorithm depending on the current needs.

Generated parser together with generated YAJCo language processor can be used to parse language sentences. The result of the parsing is an instance of the language model – instances

This work was supported by VEGA Grant No. 1/0341/13 Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.

J. Porubän (jaroslav.poruban@tuke.sk) and S. Chodarev (sergej.chodarev@tuke.sk) are with the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00, Košice, Slovakia.

[1]The tool is available at `https://code.google.com/p/yajco/`

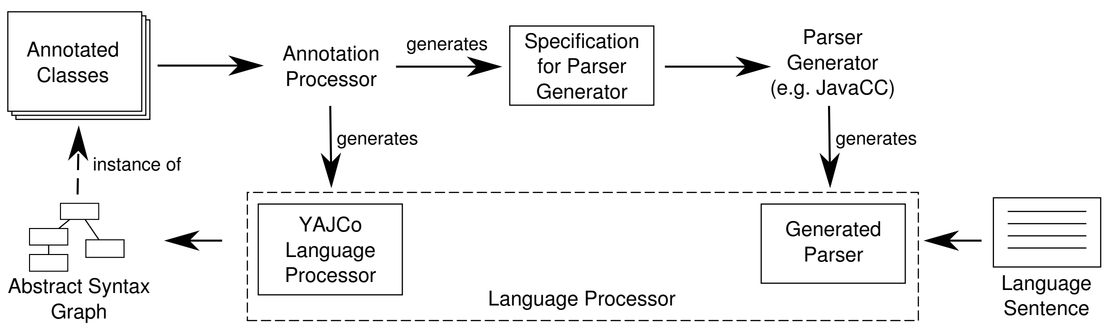


Fig. 1. YAJCo architecture

of model classes interconnected to form an abstract syntax tree corresponding to the input sentence. Moreover, YAJCo supports automatic resolution of references in the language sentences, so the output structure can be actually abstract syntax graph.

Language semantics can be defined using methods of language model classes. Alternatively it can be expressed in other classes that would traverse language model, or it can be moved into aspects using AspectJ as was shown in [6].

YAJCo can optionally generate other tools besides parser. This includes pretty-printer that is able to generate textual representation of provided object model according to the language grammar. This operation is symmetric to the parsing and makes YAJCo capable of both serialization and deserialization of objects in a specified textual form. Another tool is a visitor class that simplifies traversing the object graph.

The fact that described approach to language definition concentrates on abstract syntax also simplifies language composition [7], [8]. The language can be extended by adding concepts from other language and interconnecting them using some relations. This allows to use powerful composition mechanisms based on object-oriented and aspect-oriented programming already provided by Java and AspectJ.

III. Language Definition using YAJCo

Definition of language using YAJCo would be described based on a simple example inspired by Karel the Robot language [9]. The language consists of commands for robot movement with possibility to define subprograms. Abstract syntax of the language together with simple example of input sentence is presented in Fig. 2.

The program allows to control movements of a robot. It contains main part listing instructions for the robot, other statements (like `ITERATE` loop) and calls to subprograms (like `turnright` in the example).

For simplicity, class diagram of language model does not contain complete definition of the expression sublanguage that consists of several subclasses of the *Expression* class corresponding to different arithmetic operations. There is also only limited set of statements and instructions.

Listing 1 presents a part of the language definition. To make it short, only several classes are presented including the main class of the model – *Program*. This class has attached *@Parser* annotation, specifying name of generated parser class and lexical analyzer parameters.

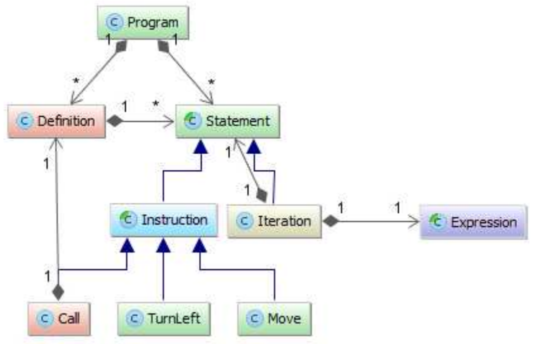
Listing 1. Fragment of the Robot language definition

```
@Parser(
  className = "yajco.robot.parser.Parser",
  skips = {@Skip("[␣\\t\\n\\r]")},
  tokens = {
    @TokenDef(name = "IDENT", regexp = "[a-zA-Z]+"),
    @TokenDef(name = "VALUE", regexp = "[0-9]+")
  }
)
class Program {
    Definition[] definitions;
    Statement[] statements;

    @Before("BEGINNING-OF-PROGRAM")
    @After("END-OF-PROGRAM")
    Program(Definition[] definitions,
            @Before("BEGIN-OF-EXECUTION")
            @After("END-OF-EXECUTION")
            @Separator(";") @Range(minOccurs = 1)
            Statement[] stmts) {
        this.definitions = definitions;
        this.statements = stmts;
    }

    void execute(Robot karel) {
        for (Statement stmt : statements)
            stmt.execute(karel);
    }
}

abstract class Statement {
    abstract void execute(Robot karel);
}

abstract class Instruction extends Statement {}

class Move implements Instruction {
    @Before("move")
    Move() {}

    void execute(Robot karel) { karel.move(); }
}
```

The use of generated parser is shown in the next listing, where a program is read from the file, parsed and its abstract representation is created. Parsed program is then executed using methods of its object model.

```
Parser parser = new Parser();
Program program = parser.parse(
       new FileReader("karel.robot"));
program.execute(karel);
```

Abstract syntax of the language consists of definition of language concepts and relations between them. In YAJCo language concepts correspond to classes and therefore, for each class corresponding non-terminal in the grammar is



```
DEFINE turnright AS
    ITERATE 3+6-2*3 TIMES
        turnleft

BEGIN-OF-EXECUTION
    turnleft;
    move;
    turnright;
    move;
    move
END-OF-EXECUTION
```

Fig. 2. Abstract syntax and example sentence of the Robot language

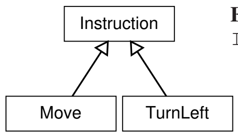
defined.

In the next sections we will discuss individual relations and properties of concepts and how YAJCo can use them to construct the parser. We will use the example language to demonstrate definition of each language property and will also provide equivalent definition in EBNF when it is possible.

A. Inheritance relation – "is-a"

Extension of a class or implementation of an interface indicates that one concepts is a special case of its parent concepts, i.e. it can be used in all places where the parent concept is expected. YAJCo automatically uses inheritance relation between classes to construct appropriate grammar rules.

For example, all instructions in the Robot language inherit from the *Instruction* class. This defines a rule for *Instruction* non-terminal with an alternative (see Fig. 3).



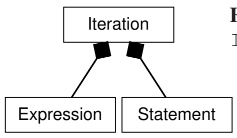
EBNF:
```
Instruction ::= Move | TurnLeft
```

Fig. 3. Definition of the inheritance relation in YAJCo and EBNF

B. Composition relation – "has-a"

A concept is composed from other concepts when description of concept instance in an input sentence includes description of its subconcepts. In the grammar it is represented by the occurrence of non-terminals of subconcepts on the right-hand side of the rule corresponding to the composed concept.

Let us take an `ITERATE` loop statement in the Robot language. Its corresponding concept *Iteration* contains an expression specifying number of iterations and a statement specifying command that would be repeated (see Fig. 4).



EBNF:
```
Iteration ::= Expression Statement
```

Fig. 4. Definition of the composition relation in YAJCo and EBNF

In object-oriented language this relation can be expressed using class fields that contain instances of subconcept classes. Not all fields, however, correspond to composition relation. Because of this, YAJCo uses parameters of class constructor for the specification of subconcepts. This also provides greater flexibility for definition of concrete syntax as it defines also ordering of subconcepts. In addition, definition of composition on constructors allows to do some preprocessing and store subconcepts in different form.

C. Keywords and symbols

The simplest form of concrete syntax annotations is a definition of keywords and symbols that must be part of concept concrete representation. These tokens can be placed before or after concept description using @*Before* and @*After* annotations associated with concept constructor. Tokens can be also placed between description of subconcepts by using the same annotations on constructor parameters (see Listing 2).

Listing 2. Example of the keywords definition

Java:
```
class Iteration extends Statement {
    int value;
    Statement statement;

    @Before("ITERATE")
    Iteration(Expression expr,
             @Before("TIMES") Statement statement) {
        this.value = expr.eval();
        this.statement = statement;
    }

    void execute(World world) {
        for(int i = 0; i < value; i++)
            statement.execute(world);
    }
}
```

EBNF:
```
Iteration ::= <ITERATE> Expression <TIMES> Statement
```

D. Composition multiplicity

A concept can contain multiple instances of subconcepts. This is automatically inferred from the use of array or one of the standard Java collection types. The multiplicity can be restricted using @*Range* annotation. There is also possibility to specify tokens that must be placed between each instance of subelement using the @*Separator* annotation (see Listing 3).

E. Alternative notations

It may be possible to describe instances of the same concept using different notations and even with different combinations

Listing 3. Example of the composition multiplicity definition
Java:
```
@Before("BEGINNING-OF-PROGRAM")
@After("END-OF-PROGRAM")
Program(Definition[] definitions,
    @Before("BEGIN-OF-EXECUTION")
    @After("END-OF-EXECUTION")
    @Separator(";") @Range(minOccurs = 1)
    Statement[] stmts) {...}
```
EBNF:
```
Program ::= <BEGINNING-OF-PROGRAM>
        Definition*
        <BEGINNING-OF-EXECUTION>
        Statement (<;> Statement)*
        <END-OF-EXECUTION>
        <END-OF-PROGRAM>
```

Listing 4. Example of alternative notations definition
Java:
```
@Before("ITERATE")
Iteration(Expression expr,
    @Before("TIMES") Statement statement) {...}

@Before("ITERATE")
Iteration(Expression expr,
    @Before("TIMES") @After("END")
    @Separator(";") @Range(minOccurs = 1)
    Statement[] statement) {...}
```
EBNF:
```
Iteration ::= <ITERATE> Expression <TIMES> Statement
Iteration ::= <ITERATE> Expression <TIMES>
              Statement (<;> Statement)* <END>
```

of subconcepts. This is allowed by the way of using multiple constructors. In the the Listing 4 extension of *Iterate* concept is presented, where alternative notation is allowed. This notation allows to specify more then one statements, but requires to end the loop with `END` keyword.

F. Operator definition

Operators represent a type of language constructs that benefits from special treatment. Otherwise they would require more complex definition of concept relations to express rules of priority and assiciativity.

In YAJCo it is possible to mark concept using @*Operator* annotation and define its priority and associativity. Annotation @*Parentheses* can be used to indicate the possibility to use parentheses to explicitly express priority.

The Robot language contains *Expression* class with several subclasses representing individual operations including addition (*Add*), subtraction, multiplication, etc. (see Listing 5).

G. References

YAJCo also supports automatic resolution of references in input sentence. A constructor parameter that expects a reference to an object defined elsewhere is marked with @*References* annotation and a field that identifies the object is marked with @*Identifier* annotation.

IV. Conclusion

The YAJCo tool allows to specify mapping between object model and language grammar. Based on the specification it

Listing 5. Example of the operator definition
```
@Parentheses(left = "(", right = ")")
abstract class Expression {
    abstract int eval();
}

class Add extends Expression {
    Expression expr1, expr2;

    @Operator(priority = 2)
    Add(Expression expr2,
        @Before("+") Expression expr2) {
        this.expr1 = expr1; this.expr2 = expr2;
    }
    int eval(){return expr1.eval() + expr2.eval();}
}
```

can generate parser and pretty-printer capable of converting text to object representation and vice versa.

It is based on the idea of the correspondence between object-models and abstract syntax. Annotations were designed for specifying concrete syntax elements missing in plain object model. The mapping also reflects several typical patterns of relation between model and grammar, including the @*Separator* for inserting a token between elements of a collection or @*Operator* for specification of priority and associativity that allow to use operations without superfluous parentheses.

We have successfully used YAJCo to develop several domain-specific languages and also for teaching model-driven software engineering approach. YAJCo is also used by DEAL method [10] for creating languages of user interfaces. On the other hand, YAJCo can be used to generate not only a parser of textual DSL, but also a user interface allowing to fill language sentences using forms [11].

References

[1] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[2] I. Halupka, J. Kollár, and E. Pietriková, “A task-driven grammar refactoring algorithm,” *Acta Polytechnica*, vol. 52, no. 5, 2012.

[3] S. Dmitriev. (2004, Nov.) Language oriented programming: The next programming paradigm. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf.

[4] S. Šimoňák, “Verification of communication protocols based on formal methods integration,” *Acta Polytechnica Hungarica*, vol. 9, no. 4, pp. 117–128, 2012.

[5] J. Porubän, M. Forgáč, M. Sabo, and M. Běhálek, “Annotation based parser generator,” *Computer Science and Information Systems*, vol. 7, no. 2, pp. 291–307, 2010.

[6] J. Porubän, M. Sabo, J. Kollár, and M. Mernik, “Abstract syntax driven language development: defining language semantics through aspects,” in *Proceedings of the International Workshop on Formalization of Modeling Languages*, ser. FML ’10. New York, NY, USA: ACM, 2010, pp. 2:1–2:5.

[7] S. Chodarev, D. Lakatoš, J. Porubän, and J. Kollár, “Language Composition based on the Composition of Concepts,” in *Informatics 2013: Proceedings of the Twelfth International Conference*. TU, 2013, pp. 133–138.

[8] D. Lakatoš and J. Porubän, “Patterns for composition of domain-specific languages,” *Journal of Computer Science and Control Systems*, vol. 6, no. 1, pp. 62–66, 2013.

[9] R. E. Pattis, *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

[10] M. Bačíková, “Deal – a method for domain analysis of graphical user interfaces,” in *Poster 2013: 17th International Student Conference on Electrical Engineering*, 2013, pp. 1–5.

[11] M. Bačíková, D. Lakatoš, and M. Nosáľ, “Automatized generating of guis for domain-specific languages,” in *CEUR Workshop Proceedings*, vol. 935, 2012, pp. 27–35.