DOI: https://doi.org/ 10.31341/jios.43.2.2
Open Access

UDC 004.4′2
Original Scientific Paper

Abstract-Syntax-Driven Development of Oberon-0 Using YAJCo

Sergej Chodarev *sergej.chodarev@tuke.sk*
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Košice, Slovakia

Michaela Bačíková *michaela.bacikova@tuke.sk*
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Košice, Slovakia

Abstract

YAJCo is a tool for the development of software languages based on an annotated language model. The model is represented by Java classes with annotations defining their mapping to concrete syntax. This approach to language definition enables the abstract syntax to be the central point of the development process, instead of concrete syntax. In this paper, a case study of Oberon-0 programming language development is presented. The study is based on the LTDA Tool Challenge and showcases details of abstract and concrete syntax definition using YAJCo, as well as the implementation of name resolution, type checking, model transformation, and code generation. The language was implemented in a modular fashion to demonstrate language extension mechanisms supported by YAJCo.

Keywords: abstract syntax, experience report, language development, language extension, Oberon-0, parser generator, YAJCo

1. Introduction

The development of computer languages, especially domain-specific languages (DSL), is an active research topic [1, 2]. Several tools have been developed that aim to support the implementation of language processors and their components including parsers, type checkers, code generators and also editing tools [3]. To compare different language development tools and formalisms, the Language Descriptions, Tools and Applications (LTDA) community defined a tool challenge[1]. Results of the challenge were later published in a special issue of the *Science of Computer Programming* journal [4].

The challenge lied in the development of a compiler for the *Oberon-0* language. *Oberon-0* is a very simple general-purpose programming language similar to Pascal. It was defined by Niklaus Wirth in his *Compiler Construction* book [5] as a subset of the Oberon programming language that is simple, but at the same time contains most of the important features of general-purpose programming languages.

In this paper, a solution to the challenge implemented using the YAJCo language processor generator [6] is presented. Main contributions of this case study are the following:

1. Implementation of the LDTA Tool Challenge using YAJCo allows comparing YAJCo with other language development tools using the already published solutions of the same challenge [4].
2. The case study demonstrates the approach of language development based on the abstract syntax definition using an object-oriented language and discusses challenges connected with this approach.
3. The study demonstrates how usual techniques for object-oriented extensibility can be used for language extension.

This paper is an extended version of our conference paper published earlier [7]. The original paper presented the incomplete implementation of the Oberon-0 without support for procedures and composed data types (arrays and records). For this paper, these missing language features were implemented as extensions to the original language definition. This allowed us to provide a more complete description of the language implementation and also to explain language extension support in greater detail.

2. YAJCo

YAJCo[2] (Yet Another Java Compiler Compiler) is an annotation based parser generator [6] — it allows to specify language syntax using annotated Java classes and generate a parser that would create instances of these classes based on the parsed sentence.

The definition of the language is derived from the classes that correspond to the abstract syntax of the language. Each class represents a language concept and corresponds to a non-terminal symbol in the grammar definition. Relations between classes (inheritance and composition) are used to construct the right-hand

1. The LTDA'2011 Tool Challenge is described at `http://ldta.info/tool.html`
2. Available at `https://github.com/kpi-tuke/yajco`

side of the grammar rules. Information that cannot be extracted from the classes, for example, details of concrete syntax, is provided in the form of Java annotations.

The generated parser is able to process text according to the language definition and create interconnected instances of the language concept classes. The composition relations between instances define a tree corresponding to the abstract syntax tree (AST). YAJCo can also automatically resolve references turning the tree into a graph.

In addition to the parser, YAJCo can generate *a visitor* — an abstract class implementing depth-first tree traversal according to the visitor design pattern [8]. Based on the visitor, YAJCo generates *a pretty-printer*, which transforms the object graph back into the textual form.

3. Implementation of Oberon-0

The Tool Challenge defines several problems to solve that can be combined in different ways. There are five compiler development tasks: (T1) parsing and pretty printing, (T2) name binding, (T3) type checking, (T4) source-to-source transformation and (T5) code generation. These tasks can be solved for five language levels. The basic level L1 defines a subset of the Oberon-0 without procedures and with only primitive types, L2 adds *for* loop and *case* statement, L3 adds procedures, L4 adds composite data types, and L5 adds pointers.

Our first paper on this topic [7] covered the implementation of the tasks for only the first two language levels. For this paper, all tasks (T1–5) were implemented for the first four language levels (L1–4). The implementation of the language[3] consists of the following components:

1. Metamodel — language abstract syntax definition in the form of annotated Java classes.

2. Model analysis and transformation modules: name resolver, type checker, transformer and code generator.

YAJCo is used to generate parser, pretty-printer, and visitor from the metamodel. The visitor is then extended by all components that implement the model analysis and transformation because they need to traverse the AST for their functionality.

The development process was based on test-driven development [9]. The metamodel was defined incrementally by adding more language constructs in each step and immediately testing the generated parser. Small Oberon-0 programs were defined as test cases to ensure correct processing of the language constructs. The

3. Source code of the implementation can be downloaded at `https://git.kpi.fei.tuke.sk/sergej.chodarev/yajco-oberon0`

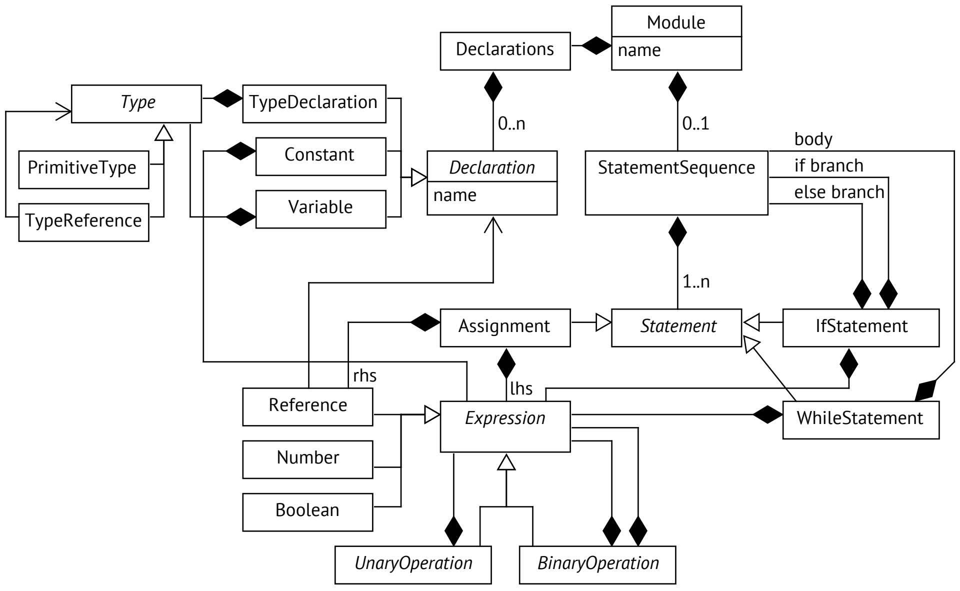

Figure 1: Metamodel representing the abstract syntax of the Oberon-0 L1 (concrete operators are omitted)

analysis and transformation modules were also implemented incrementally with added language concepts.

4. Syntax Definition

You can see the diagram of classes that form the abstract syntax of the Oberon-0 in Figure 1. For simplicity, only classes that are part of the language level L1 are displayed and operators are excluded.

The root of the language is the *Module* concept. The *Module* contains different kinds of declarations and a sequence of statements. Declarations and statements may contain expressions that include references to variables or constants and various binary and unary operators, such as addition (+) or equality testing (=).

All concepts of the language are represented by Java classes. We recognize three types of relations between concepts:

1. Inheritance (is-a) — relation between an abstract concept and more concrete concepts. For example, *Variable* is a concrete type of *Declaration*.

2. Composition (has-a) — relation between a language concept and its parts. For example, a *Variable* declaration contains its *Type*.

3. Reference — relation where a concept does not directly contain another concept, but only a reference to it. For example, a *TypeReference* represents

cases, where a type is specified by name and not declared directly in-place. Therefore, it contains a reference to the real *Type* defined elsewhere.

Most of the concepts defined in the abstract syntax correspond directly to non-terminals in the grammar definition of the Oberon-0 [5]. Abstract syntax, however, does not follow the concrete syntax definition in all details. First of all, abstract syntax contains explicit references to other concepts, instead of referencing them implicitly using identifiers. It also omits non-terminals that represent only detail of concrete notation, for example, *IdentList* used to group identifiers with the same type.

On the other hand, the abstract syntax contains concepts that are not directly present in the grammar, like *PrimitiveType* or *Boolean*. Primitive type names (`BOOLEAN` and `INTEGER`) and boolean values (`TRUE` and `FALSE`) are defined in Oberon-0 as built-in identifiers instead of keywords, so they are not included in the grammar. They are, however, concepts of the language, so they are part of the abstract syntax definition.

Definition of the abstract syntax using Java classes corresponds to domain modeling and it requires to identify language concepts and express relations between them in the early stages of the language development.

4.1 Concrete Syntax Definition

Listing 1 provides an example of the language concept definition. It defines the *Module* concept with two allowed concrete syntax forms: with statements and without them (a corresponding grammar fragment in the extended BNF is provided in Listing 2). Each syntax form is defined using a constructor of the class. Composition relations between classes are derived from the types of constructor parameters.

Keywords that are used in the language concept concrete syntax are defined using @*Before* and @*After* annotations. Notice that additional syntax constraints can be checked inside the constructor body, for example, Oberon-0 requires module name to be repeated at the end of the definition after the `END` keyword.

YAJCo also provides special support for most common syntactic patterns like sequences with separator symbols or infix operators. For example, the *StatementSequence* definition in Listing 3 uses the @*Separator* and @*Range* annotations to specify that there must be at least one statement and they must be separated by a semicolon.

Using the @*Operator* annotation it is possible to define the priority level and associativity of the operator as shown in Listing 4. To support enclosing operators in parentheses, the base class *Expression* is marked with the @*Parentheses* annotation. YAJCo uses this information to automatically generate the corresponding grammar rules without the need to modify the metamodel structure to support these concrete syntax features.

```
public class Module {
    private String name;
    private Declarations declarations;
    private StatementSequence statements;

    @Before("MODULE") @After(".")
    public Module(
            @After(";") String name,
            Declarations declarations,
            @Token("name") @Before("END") String nameRepeated) {
        if (!name.equals(nameRepeated))
            throw new RuntimeException("...");
        this.name = name;
        this.declarations = declarations;
    }

    @Before("MODULE") @After(".")
    public Module(
            @After(";") String name,
            Declarations declarations,
            @Before("BEGIN") StatementSequence statements,
            @Token("name") @Before("END") String nameRepeated) {
        this(name, declarations, nameRepeated);
        this.statements = statements;
    }
    ...
```

Listing 1: Definition of the Module concept

```
module = "MODULE" ident ";" declarations
       ["BEGIN" StatementSequence] "END" ident "." .
```

Listing 2: Definition of the module in the EBNF form

```
public class StatementSequence extends ArrayList<Statement> {
    public StatementSequence(
            @Separator(";") @Range(minOccurs = 1)
            List<Statement> statements) {
        addAll(statements);
    }
    ...
```

Listing 3: Definition of the addition operator

```
public class Add extends BinaryOperation {
    @Operator(priority = 2, associativity = LEFT)
    public Add(Expression left, @Before(”+”) Expression right) {
        super(left, right);
    }

    @Override
    public Type getType() { return INTEGER; }
}
```

Listing 4: Definition of the addition operator

```
public class IfStatement extends Statement {
    ...
    @Before(”IF”) @After(”END”)
    public IfStatement(
            Expression condition,
            @Before(”THEN”) StatementSequence thenBranch,
            ElsifFragment elsif) {
        this.condition = condition;
        this.thenBranch = thenBranch;
        this.elseBranch = StatementSequence.of(
            elsif.getIfStatement());
    }
    ...
}
```

Listing 5: One of constructors of the *IfStatemetns* class

4.2 Helper Classes for Concrete Syntax Mapping

In some cases, the relation between the abstract and concrete syntax is more complex and cannot be expressed using constructor parameters and annotations. For example, Oberon-0 grammar defines an `ELSIF` keyword used as a part of the *if* statement. It is actually a shortcut that allows a more convenient nesting of the *if* statement inside the *else* branch of the previous statement. As this construct is just “syntactic sugar” it is not required to represent it in the abstract syntax graph using a special kind of object.

In such cases, special classes need to be introduced to represent concrete syntax features instead of abstract syntax concepts. In the case of the `ELSIF` keyword, an *ElsifFragment* class was defined with the syntax definition of this element. In the corresponding constructor of the *IfStatement* class, the nested *if* statement is extracted and stored as a single statement of the *else* branch (see Listing 5). The *ElsifFragment* class, therefore, influences the grammar and parser of the language, but its instances are not stored in the abstract syntax graph.

```
declarations = ["CONST" {ident "=" expression ";"}]
               ["TYPE" {ident "=" type ";"}]
               ["VAR" {IdentList ":" type ";"}].
```

Listing 6: Concrete syntax of Oberon-0 declarations (without procedures)

```
public class ConstantDeclarations extends ArrayList<Constant> {
    public ConstantDeclarations() {}

    @Before("CONST")
    public ConstantDeclarations(
            @Range(minOccurs = 1) List<Constant> declarations) {
        addAll(declarations);
    }

    public List<Constant> getDeclarations() {
        return this;
    }
}
```

Listing 7: Definition of constant declarations section

Another example of classes specific to concrete syntax is provided by the declarations section of the module definition. From the point of view of the abstract syntax model, *Declarations* class is a simple collection of *Declaration* objects, where *Declaration* is an abstract base class for all kinds of declarations (see class diagram in Fig. 1). In the concrete syntax, however, declarations contain separate lists of constants, types, and variables that cannot be intermixed (see grammar rule in Listing 6). In addition, multiple variables can be grouped together into a single declaration if they are of the same type.

To map the abstract syntax to the concrete syntax, several helper classes were used. Each list from the declarations section is represented using a class that is actually a list with custom constructors (see Listing 7 for an example). Details of the declaration syntax are specified in a class representing list item (e.g. *Constant*).

For variables grouped by their type, an additional helper class was defined — *VariablesGroup* with a constructor that defines the concrete syntax of the grouping. The constructor of the *VariableDeclarations* class then merges these groups into a single list of *Variable* objects.

Finally, the constructor of the *Declarations* class merges all lists of declarations into a single hash map, where they are accessible by name. Only this representation is actually part of the abstract syntax. Instances of the helper classes are not needed for further processing so they are not stored to avoid duplication.

4.3 Pretty-Printer

Pretty-printer generator uses the same grammar extracted from the class structure and constructors. In addition, it requires *get methods* for all constructor parameters to be implemented. This means that the objects must be able to return values for these parameters even if they do not directly correspond to fields of the class. For example, you can see such get method also in Listing 7, where it is needed only for the pretty-printer.

The need to have these get methods requires some additional work in cases were concrete syntax does not match directly to the abstract syntax and helper classes are used. If their instances are not stored in the model, they should be recreated in the get methods using an inverse transformation. For example, in the case of the *Declarations* class, separate get methods do filtering of the declarations list based on the declaration's type.

5. Model Analysis and Transformation

Attaching semantics to the language metamodel represented by Java classes can be done in different ways. The most obvious way is to implement semantic actions as methods of the metamodel classes. They can be separated from the metamodel definition using aspect-oriented programming techniques [10] or can be implemented using a visitor design pattern.

In our case, most of the model analysis and transformation operations were implemented based on the *Visitor* class generated by YAJCo. The *Visitor* class implements the visitor design pattern for the classes that are part of the syntax definition and provides default implementations for all methods allowing full depth-first traversal through the abstract syntax tree, so only methods that would do some operations need to be overridden. In addition, all of the *visit* methods of the class accept an additional parameter allowing to pass some context.

5.1 Name Analysis

YAJCo provides its own mechanism for automatic name resolution [11]. However, because of the limitations of its current implementation (it does not support inheritance hierarchy of referenced classes), it was required to implement a custom name resolution module.

The *NameResolver* class extends the visitor and passes the *SymbolTable* object, as a context between visit methods. As can be seen in Listing 8, referenced names are first looked up in the list of built-in constants (like `TRUE` and `FALSE`) and then in the symbol table. The found declaration is stored in the field of the corresponding reference object. In a case where the declaration is missing, an error message is added into a list of errors that would be reported to the user.

```
public class NamesResolver extends Visitor<SymbolTable> {
    @Override
    protected void visitReference(Reference reference,
                                  SymbolTable declarations) {
        String name = reference.getName();
        Constant constant = checkBuiltinConstants(name);
        if (constant != null) {
            reference.setDeclaration(constant);
            return;
        }
        Declaration declaration = declarations.get(name);
        if (declaration == null) {
            errors.add(...);
        }
        reference.setDeclaration(declaration);
    }
    ...
```

Listing 8: Fragment of the *NameResolver* class

The *NameResolver* also resolves built-in named constants (`TRUE` and `FALSE`), type names, procedure names and parameters.

5.2 Type Checking

The type of expressions and declarations is defined as a part of the language model. Therefore, each language concept with a type provides a *getType()* method (see for example Listing 4). Oberon-0 does not have any polymorphic operations, so definition of the *getType()* methods is straightforward — in case of operators they just return a constant value based on the type of the operator, for a variable it returns the declared type of the variable, and for a constant it returns the type of the initialization expression bound to the constant.

The type checker, therefore, checks types of operands in expressions and control structures (for example, the condition of the *if* statement must be of type *boolean*). It also checks if the type of the expression matches the type of the variable in variable assignments. Found type errors are collected into a list representing the result of the type checking.

5.3 Source-to-Source Transformation

According to the definition of the challenge, a source-to-source transformation was implemented for "de-sugaring" of the *case* and *for* statements added in the second language level L2 (see section 6) and for "lifting" nested methods in the level L3.

In both cases, the transformation was done using the visitor pattern and the abstract syntax graph was modified in-place.

De-sugaring case and for statements. During the processing, they were transformed into the concepts of the base language level L1. In this case, the visitor analyzes all instances of the *StatementSequence* class found in the abstract syntax tree. If any of the transformed statement types were found, a transformation method is applied to replace them with corresponding concepts from the L1 language level. The transformation methods just create the instances of the replacement objects and initialize them according to the translation schema.

Procedures lifting. In this case, all procedures were checked if they contain nested procedure declarations. In such a case, the nested *Procedure* objects were moved to the main *Module* declaration and their names are mangled to contain the name of the parent procedures as a prefix. Procedure calls do not need modification, because they contain a reference to the actual *Procedure* declaration with the updated name.

5.4 Code Generation

The code generation module realizes the translation of the Oberon-0 into ANSI C. The languages provide similar features, so the translation is implemented using a simple visitor with a *PrintWriter* instance as a context object used to emit the generated code.

6. Language Extension

As was mentioned earlier, we have implemented four language levels defined by the Tool Challenge:

- L1 — base version of the language without procedures and with only simple types,
- L2 — previous level extended with Pascal-like `FOR` and `CASE` statements,
- L3 — previous level extended with declarations and calls of procedures,
- L4 — previous level extended with `ARRAY` and `RECORD` types.

To showcase language extension support provided by YAJCo we have implemented all relevant tasks for each language level separately. Therefore, in each step, we have implemented a complete translator from Oberon-0 to C with type checking, name resolution, and tree transformation. Therefore, we have extended not only syntax definition but also the semantics of the language.

6.1 Syntax Extension

YAJCo has full support for language composition [12], so an extension can be implemented straightforwardly by adding classes representing the new language

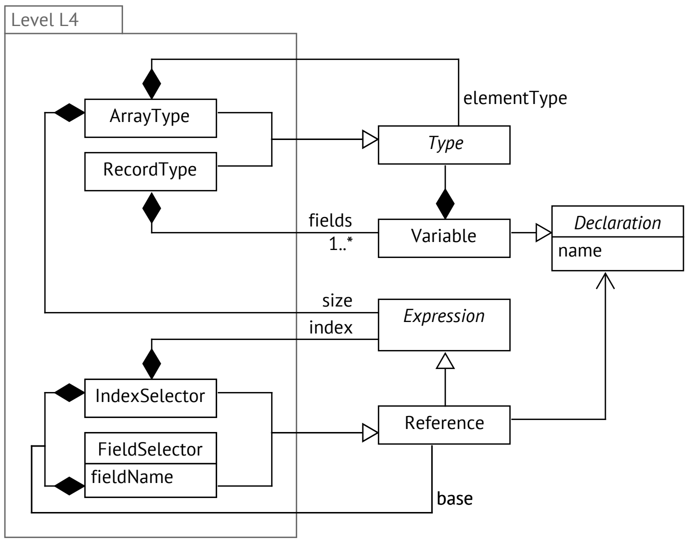

Figure 2: Classes representing the language extension L4 and related classes from the base level

elements and extending class from the base language. This addition is automatically detected by YAJCo and new concepts are incorporated into the language definition, resulting in an extended parser and pretty-printer.

For example, new classes representing the `FOR` and `CASE` statements are defined as subclasses of the *Statement* class and are automatically added to the grammar, generated parser, and pretty-printer. The addition of the procedure call statement in the level L3 was similar.

As you can see in Figure 2, declarations of array and record types in the level L4 extended the *Type* class from the base language level. This means that they can be used in all places, where a type is expected, including variable declaration and declaration of a new type. Similarly, new subtypes of the *Reference* class was defined to support array indexes and record field selectors after a variable name. These types are recursive — they contain a base *Reference* and therefore can be nested.

Declaration of procedures was less trivial. Since the original implementation of the *Declarations* class explicitly listed each type of supported declarations (types, constants, and variables) in its constructor, it was not enough to extend the *Declaration* class. A new class was created — *DeclarationsWithProcedures* extending *Declarations* and defining a constructor accepting procedures in addition to previously supported concepts. This new class was automatically detected by YAJCo and the grammar was extended with an alternative corresponding to it.

6.2 Semantics Extension

As was mentioned earlier, the semantics of a language can be implemented directly in methods of the abstract syntax classes, or it can be separated into separate classes usually extending the *Visitor* class generated by YAJCo. Extension of the language is possible in both cases. In the first one, new language elements are added with the corresponding semantic methods, or semantics of existing elements is modified using aspect-oriented programming [12]. In the second case, new processing classes are defined using the usual object-oriented extension mechanism and overriding the necessary *visit* methods. The abstract *Visitor* class is automatically updated by YAJCo to contain default implementations of *visit* methods for all detected language elements that implement full traversal of the tree.

We have used visitor classes to implement the semantics of the Oberon-0. For example, the type checking system for the level L1 was implemented in the *TypeChecker* class using the visitor design pattern. For each of the next levels, we have implemented a new type checker that extends the previous one and override needed methods. Name resolver, graph transformer, and code generator were also extended where it was needed.

The most complex change was the extension of the names resolution system to support the concept of nested namespaces in the level L3. The original names resolver used the *Declarations* class directly to lookup objects for names. We have refactored it by introducing a new *SymbolTable* interface implemented by *Declarations* and using this interface in the *NamesResolver*. This allowed to simply replace *Declarations* with *StackedDeclarations* for processing level L3. Because of the common interface, the code inherited from the base class can work correctly with nested namespaces.

The introduction of interfaces and changing the visibility of some methods from *private* to *protected* was required in several other cases. This simplified the extension of existing classes and allowed us to override and extend parts of their functionality while reusing the rest. This illustrates that language extension in the case of YAJCo is equivalent to the usual object-oriented extension. It uses the same techniques, requires some adherence to object-oriented design best practices and also benefits from upfront design with respect to future extensions.

7. Observations

YAJCo allows defining languages using usual object-oriented concepts of classes, inheritance, compositions, and association. The classes are written directly by the language developer and not generated from the grammar definition. This means that the developer may choose the best representation of the abstract syntax for further processing.

The approach based on the abstract syntax also shifts the focus of language development to the more abstract specification of the conceptual model of the language instead of its concrete syntax. A language developer is required to pay attention to conceptual relations between language elements before specifying concrete syntax.

Implementation of model analysis and transformation using object-oriented code allows applying usual code abstraction and modularization techniques, such as the introduction of methods for repeating code and extension using inheritance. Language implementation, therefore, benefits from the well-established object-oriented design principles that simplify its extension and evolution.

The generated visitor allows implementing tree traversal in a very straightforward and concise way. It supports language extension because the addition or modification of language concepts that are not directly processed by the visitor subclasses would not alter their functionality — the processor would simply inherit the default implementation of the corresponding *visit* methods from the base class.

Another important advantage of expressing language concepts directly using Java classes lies in the native support of integrated development environments (IDE). It is possible to use rich refactoring and code generation features provided by modern IDEs to modify and extend the language definition and all processing code.

This case study also exposed some deficiencies in the current implementation of the YAJCo tool. For example, optional constructor arguments are not directly supported. The same effect can be achieved using multiple constructors, however, direct support of the *Optional* type from Java 8 would be useful.

As mentioned earlier, the built-in name resolution mechanism does not support referencing different subclasses of a base class. An important missing feature is the ability to map objects from the abstract syntax graph to source code positions for better error reporting.

Helper classes used to represent complex relations between concrete and abstract syntax can also complicate the language definition. In the future, it is desirable to identify and support more common transformations similarly to the currently supported sequences and operators.

8. Related Work

Domain-specific languages are successfully used in different areas, for example, application logging [13], acceptance tests [14], graphic shape description [15], expert systems [16], text analysis [17], or automatic assessment of students [18]. Source code annotations can also be considered a DSLs [19]. It was shown that DSLs improve program comprehension compared to general-purpose languages [20]. Therefore tools and methods for the development of DSLs are active research topics.

Several other language development tools were used to solve the tasks of the LDTA Tool Challenge. Part of them uses attribute grammars as a formalism allowing analysis and transformation of AST, including Silver [21] and CoCoCo [22] that use a functional language for the definition of attributes.

JustAdd [23] uses object-oriented attribute grammars, where the abstract syntax is represented by classes, but parser specification and its mapping to the abstract syntax is defined separately using the Beaver parser generator. In our case, the Beaver can also be used as an underlying parser generator, but the specification for it is generated automatically.

Kiama [24] is another tool based on attributed grammars. It uses Scala type classes for representing the abstract syntax and several DSLs embedded in Scala to define other properties of a language.

Simpl DSL toolkit [25] is also based on Scala and uses the case classes. They are, however, generated automatically based on the grammar specification. Analysis and transformation of the AST are handled by a usual Scala code. In this sense, this tool is similar to YAJCo with an important difference in the fact that Simpl uses concrete syntax specification as the main artifact instead of the abstract syntax.

Rascal [26] is a specialized language for meta-programming. It provides built-in constructs for syntax definition, and strategic traversal and rewriting of trees. The challenge was also solved using the Objective Caml programming language [27] that by default provides tools sufficient for language processing, including parser combinators and “type-driven” transformers.

9. Conclusion

The paper presents an experience report of implementing the Oberon-0 language using the YAJCo parser generator. This tool supports the development driven by the abstract syntax definition instead of the concrete syntax. Together with the use of the usual object-oriented programming approach and Java language it provides possibilities to utilize the well-known software engineering tools and approaches also for language development and make this area more approachable for developers.

We plan to use the experience from this experiment to eliminate the identified deficiencies of the YAJCo tool. The design and implementation of additional patterns for concrete syntax specification is the main goal of our future research in this area. For example, support for the optional parts of a language concept could be easily added. Grouping of elements with the same properties is also a common pattern, that can be supported directly using a special annotation.

Another area of improvement is tool support. We plan to support the generation of a complete developed environment from the language definition.

Acknowledgment

This work was supported by projects VEGA No. 1/0762/19: "Interactive pattern-driven language development" and FEI TUKE Grant no. FEI-2018-57 "Representation of object states in a program facilitating its comprehension".

References

[1] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, dec 2005. doi: 10.1145/1118890.1118892

[2] T. Kosar, S. Bohra, and M. Mernik, "Domain-Specific Languages: A Systematic Mapping Study," *Information and Software Technology*, vol. 71, pp. 77–91, mar 2016. doi: 10.1016/J.INFSOF.2015.11.001

[3] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning, "Evaluating and comparing language workbenches," *Computer Languages, Systems & Structures*, vol. 44, pp. 24–47, dec 2015. doi: 10.1016/j.cl.2015.08.007

[4] M. van den Brand, "Introduction—the LDTA tool challenge," *Science of Computer Programming*, vol. 114, pp. 1–6, dec 2015. doi: 10.1016/j.scico.2015.10.015

[5] N. Wirth, *Compiler construction*. Addison-Wesley, 1996. ISBN ISBN 0-201-40353-6

[6] J. Porubän, M. Forgáč, M. Sabo, and M. Běhálek, "Annotation based parser generator," *Computer Science and Information Systems (ComSIS)*, vol. 7, no. 2, pp. 291–307, 2010. doi: 10.2298/CSIS1002291P

[7] S. Chodarev and M. Bacikova, "Development of Oberon-0 using YAJCo," in *2017 IEEE 14th International Scientific Conference on Informatics*. IEEE, nov 2017. doi: 10.1109/INFORMATICS.2017.8327233. ISBN 978-1-5386-0888-3 pp. 122–127.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2

[9] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[10] J. Porubän, M. Sabo, J. Kollár, and M. Mernik, "Abstract syntax driven

language development," in *Proceedings of the International Workshop on Formalization of Modeling Languages - FML '10*. ACM Press, 2010. doi: 10.1145/1943397.1943399. ISBN 9781450305327 pp. 1–5.

[11] D. Lakatoš, J. Porubän, and M. Bačíková, "Declarative specification of references in DSLs," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2013. ISBN 9781467344715 pp. 1527–1534.

[12] S. Chodarev, D. Lakatoš, J. Porubän, and J. Kollár, "Abstract syntax driven approach for language composition," *Open Computer Science*, vol. 4, no. 3, p. 160, jan 2014. doi: 10.2478/s13537-014-0211-8

[13] S. Zawoad, M. Mernik, and R. Hasan, "Towards building a forensics aware language for secure logging," *Computer Science and Information Systems (ComSIS)*, vol. 11, no. 4, pp. 1291–1314, 2014. doi: 10.2298/CSIS131201051Z

[14] T. Straszak and M. Śmialek, "Model-driven acceptance test automation based on use cases," *Computer Science and Information Systems (ComSIS)*, vol. 12, no. 2, pp. 707–728, 2015. doi: 10.2298/CSIS141217033S

[15] J. Kollár and M. Spišiak, "Direction vector grammar," in *2015 IEEE 13th International Scientific Conference on Informatics*. IEEE, nov 2015. doi: 10.1109/Informatics.2015.7377824. ISBN 978-1-4673-9867-1 pp. 151–155.

[16] M. Woźniak, D. Polap, C. Napoli, and E. Tramontana, "Graphic object feature extraction system based on Cuckoo Search Algorithm," *Expert Systems with Applications*, vol. 66, pp. 20–31, dec 2016. doi: 10.1016/j.eswa.2016.08.068

[17] M. Sičák and J. Kollár, "Supercombinator set construction from a context-free representation of text," in *Federated Conference on Computer Science and Information Systems (FedCSIS 2016)*, oct 2016. doi: 10.15439/2016F334. ISBN 9788360810903 pp. 503–512.

[18] E. Pietriková, J. Juhár, and J. Šťastná, "Towards automated assessment in game-creative programming courses," in *International Conference on Emerging eLearning Technologies and Applications (ICETA 2015)*. IEEE, nov 2015. doi: 10.1109/ICETA.2015.7558505. ISBN 978-1-4673-8534-3 pp. 1–6.

[19] M. Nosáľ, M. Sulír, and J. Juhár, "Language composition using source code annotations," *Computer Science and Information Systems (ComSIS)*, vol. 13, no. 3, pp. 707–729, 2016. doi: 10.2298/CSIS160114024N

[20] T. Kosar, N. Oliveira, M. Mernik, V. J. M. Pereira, M. Črepinšek, D. Da

Cruz, and R. P. Henriques, “Comparing general-purpose and domain-specific languages: An empirical study,” *Computer Science and Information Systems (ComSIS)*, vol. 7, no. 2, pp. 247–264, 2010. doi: 10.2298/CSIS1002247K

[21] T. Kaminski and E. Van Wyk, “A modular specification of Oberon0 using the Silver attribute grammar system,” *Science of Computer Programming*, vol. 114, pp. 33–44, 2015. doi: 10.1016/j.scico.2015.10.009

[22] M. Viera and S. D. Swierstra, “Compositional compiler construction: Oberon0,” *Science of Computer Programming*, vol. 114, pp. 45–56, 2015. doi: 10.1016/j.scico.2015.10.008

[23] N. Fors and G. Hedin, “A JastAdd implementation of Oberon-0,” *Science of Computer Programming*, vol. 114, pp. 74–84, 2015. doi: 10.1016/j.scico.2015.02.002

[24] A. M. Sloane and M. Roberts, “Oberon-0 in Kiama,” *Science of Computer Programming*, vol. 114, pp. 20–32, 2015. doi: 10.1016/j.scico.2015.10.010

[25] M. Freudenthal, “Simpl DSL toolkit,” *Science of Computer Programming*, vol. 114, pp. 85–91, dec 2015. doi: 10.1016/j.scico.2014.11.018

[26] B. Basten, J. Van Den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. Van Der Ploeg, T. Van Der Storm, and J. Vinju, “Modular language implementation in Rascal - Experience report,” *Science of Computer Programming*, vol. 114, pp. 7–19, 2015. doi: 10.1016/j.scico.2015.11.003

[27] D. Boulytchev, “Combinators and type-driven transformers in Objective Caml,” *Science of Computer Programming*, vol. 114, no. 13, pp. 57–73, 2015. doi: 10.1016/j.scico.2015.07.008