Computer Science and Information Systems 14(3):939–958 DOI: https://doi.org/10.2298/CSIS170116036C

Development of Custom Notation for XML-based Language: a Model-Driven Approach

Sergej Chodarev and Jaroslav Porubän

Technical University of Košice, Department of Computers and Informatics,
Letná 9, Košice, Slovakia
{sergej.chodarev, jaroslav.poruban}@tuke.sk

Abstract. In spite of its popularity, XML provides poor user experience and a lot of domain-specific languages can be improved by introducing custom, more human-friendly notation. This paper presents an approach for design and development of the custom notation for existing XML-based language together with a translator between the new notation and XML. The approach supports iterative design of the language concrete syntax, allowing its modification based on users feedback. The translator is developed using a model-driven approach. It is based on explicit representation of language abstract syntax (metamodel) that can be augmented with mappings to both XML and the custom notation. We provide recommendations for application of the approach and demonstrate them on a case study of a language for definition of graphs.

Keywords: domain-specific languages, human-computer interaction, iterative design, model-driven development, translator, XML.

1. Introduction

XML is very common and easy to parse generic language. It is well supported by existing tools and technologies and therefore it is a popular basis for domain-specific languages (DSLs). While XML is appropriate choice in many cases, especially for program-to-program communication, it is not well suited for cases, where humans need to manipulate documents. Although they are able to create, modify and read XML documents, it is not a pleasurable experience, because of uniformity and syntactic noise that makes it difficult to find useful information visually [28].

While a more appropriate syntax can be chosen for the development of new languages, a lot of languages was already implemented based on XML and their reimplementation would be complicated and time-consuming. One of the possible ways to solve this problem is to develop a translator that would read documents written in a specialized human-friendly notation and output them in the XML for further processing using existing tools. Ideally, the new notation would be specifically tailored to the domain of the language as is usual for DSLs [22].

In this paper we present an approach that supports iterative design of the notation. It is possible to evaluate the notation by automatically converting samples of existing documents from XML in each iteration. This makes it easier to experiment with different syntax alternatives and choose the most appropriate one.

To make such iterative process possible, we propose to use model-driven approach. This means that instead of traditional language development approach driven by *concrete*

syntax definition, we need to base development of the translator on *the abstract syntax* that is common for all notations of the language [16]. Abstract syntax should be expressed in a format that can be easily augmented with the definition of both notations and allow automatic generation of corresponding language processors.

For example, Java classes representing the structure of an XML-based language can be generated automatically from the XML Schema using JAXB[1]. The generated classes are already annotated in a way that allows automatic marshalling and unmarshalling their instances in the XML form. Additional annotations can be added to the classes that define their mapping to a different textual notation. In the next step an annotation based parser generator, like YAJCo [30], can be used to generate a parser and pretty-printer for the new notation. Connecting them with the XML marshaller and unmarshaller one would get a complete translator from the custom human-friendly notation to the original XML-based and back.

Rest of the paper is structured based on the main topics and contributions of the paper, that are the following:

1. The process of *iterative design* of new notation for existing language (Section 2). This process is in contrast with traditional approach, where concrete syntax is completely defined before the development of language processor.
2. The approach to language translator development that is based on explicit representation of language *abstract syntax* in a format that allows attaching definitions of different concrete notations (Section 3). This allows to develop a round-trip translator based on the specification of the abstract syntax.
3. Demonstration of the approach on *a case study* of a language for specification of graphs (Section 4). The case study shows possible challenges of the approach and can be used as a guide to develop similar translators.
4. Summary of *recommendations* for application of the approach that was generalized from the case study (Section 5).

The approach was originally presented in our conference paper [6]. In this extended version of the paper larger emphasis was given to the iterative process of notation design, which is now explained in greater detail.

This paper also presents completely new case study. In the previous paper a language for graphical user interface specification was used (examples from the original case study are provided in Appendix A). While the language demonstrated that the new notation could be much shorter and compendious compared to XML, it did not have complete specification in a form of XML Schema making the development more complicated. The new case study uses the GraphML language with proper XML Schema. The study also includes discussion of alternative solutions and describes testing of the translator.

In addition, the recommendations was extracted from both case studies, that summarize most important points in a tool-neutral form.

2. Iterative Process of Language Notation Design

Notation of a formal language defines a way how it is presented to its users and how they interact with code written in the language. Therefore, notation is a user interface

[1] Java Architecture for XML Binding: https://docs.oracle.com/javase/tutorial/jaxb/

of the language and design of the notation should follow the principles of user interface design [2]. This means it requires iterative evaluation of the design and its modification based on the evaluation results [25]. Development of the translator between the new notation and the original one should follow the same iterations, so the translator would allow to test the design in conditions similar to real life.

On the other hand, classical approach to language development [1] assumes that concrete syntax of the language is designed upfront. A complete specification of the grammar is then augmented with semantic actions and processed to generate a parser. Therefore, changes in the syntax often require modification of semantic rules, making the process laborious.

We propose an alternative process for development of the custom notation for existing XML-based languages together with the round-trip translator:

1. Extract the language abstract syntax from the XML Schema.
2. Augment the abstract syntax with initial definition of the new concrete syntax.
3. Generate a pretty-printer based on the definition.
4. Convert examples of existing XML documents to the new notation.
5. Evaluate the new notation on examples of converted documents.
6. If the notation is not satisfactory, modify the concrete syntax definition and go back to the step 3.
7. If the notation is satisfactory, complete the syntax definition and generate the parser.

In the first step, the central piece of the translator — language abstract syntax is defined based on the existing language definition. The first iteration then starts with the design of the initial version of the notation. Definition of the notation must support generation of a pretty-printer based on it. More detailed discussion of the implementation is provided in the Section 3.

A set of existing documents in the XML-based notation is converted to the new one and manually evaluated. Based on the results of the evaluation, the notation is either redesigned and reimplemented based on the feedback, or it is finalized to obtain full round-trip translator between the notations.

This process allows to easily use existing documents for testing the new notation instead of some artificial examples. Complete real-life documents in the new notation can be generated automatically immediately after the definition of the syntax has changed. This allows very fast evaluation and modification cycles, so problems in the notation can be spotted and resolved, even if they occur only in complex documents.

The evaluation can be done in different ways depending on the needs of the project. In the simplest case it consists of visual checking of comprehensibility of converted documents. Usual usability testing methods can be used as well. This includes testing with potential users of the language, in which they would be given realistic tasks. For example, *discount usability evaluation method* can be successfully applied to software languages [19]. Quantitative evaluation methods can be used as well [2], although they require larger number of participants to obtain statistically significant results.

This approach also provides a simple method for testing correctness of the developed translator, i.e. that no information is lost or corrupted during the translation. A set of example XML documents can be automatically converted to the new notation and then back to the XML. Result of the conversion can be compared with the original XML documents

to reveal missing support for some language features or other errors. If the translator is correct, no data is lost and documents are identical (except of differences in formatting that can be removed using normalization before the comparison).

The approach is not limited to XML-based languages. With some modifications it can be used for development of alternative notation for any software language.

3. Model-driven Development of Language Translator

To support described process, it is needed to use approach similar to model-driven software development [33], where the development of the language translator is driven by the model of the language — metamodel[2]. The metamodel defines language concepts with their properties and relations to other concepts. Definition of the structure is annotated with additional information about concrete syntax of the language that needs to be translated.

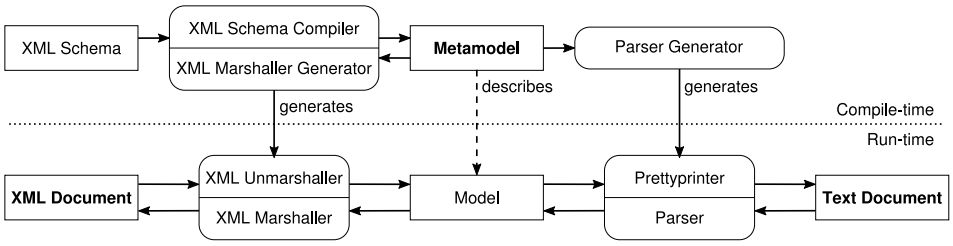

Fig. 1. Model-driven language translator development (arrows represent data-flow)

Figure 1 shows the whole architecture of the model-driven language translator development in the case of translating XML to textual notation and vice versa. The metamodel augmented with definition of concrete notations is the central element. It is used as an input to generate parser and pretty-printer for both the textual notation (using parser generator) and XML (using XML marshaller generator). The generated tools can be connected into a pipeline that handles the translation of one notation to the other with the internal representation of the model (defined by the metamodel) as an intermediate format.

What is important, the first version of the metamodel itself can be retrieved from the existing description of XML-based language — XML Schema. This allows to significantly shorten the development process, because large part of the language definition — its abstract syntax specification — is derived automatically.

This style of development also follows the "Single Point of Truth" principle [31], because the structure of the language is defined only once and its mappings to concrete notations are attached to it. Therefore, it is easier to keep concrete syntax definitions in sync.

In the case of evolution of the language, the changes should be expressed in the metamodel, so other artifacts, including the XML Schema, could be updated automatically. If

[2] If we consider documents written in a language to be *models*, then a model of the language itself is *a metamodel*.

it is not possible, it would be required to manually synchronize changes of the language with the metamodel definition.

The described approach does not depend on concrete tools. It, however, requires an XML marshaller and a parser/pretty-printer generator that both use the same format for the metamodel specification. In the case study in Section 4 Java classes are used to represent the metamodel. They are augmented using annotations, JAXB is used as an XML marshaller and YAJCo as a parser generator. Alternative solution can use Ecore from EMF [34] to represent the metamodel and Xtext [8] as a parser generator.

The approach is based on interconnecting different technological spaces (TS) [18]. Original language and its infrastructure is defined in the XML technological space, but for definition of the new textual notation it is appropriate to use the programming languages syntax TS. Both spaces are interconnected using the abstract syntax definition, that by itself can be placed in a different technological space. In our case it is object-oriented programming TS, but it can be Model-Driven Architecture TS if different representation of metamodel would be chosen (e.g. Ecore). The choice of technological space would substantially influence approaches and technologies used to solve the task of language translation. For example, while in the MDA TS some model transformation language (e.g. ATL [13]) would be used, in OOP TS the same task would be solved using methods of the classes representing metamodel or using the Visitor design pattern.

From this point of view, presented case study also demonstrates that object-oriented programming language like Java can be successfully used as a format for abstract syntax description, provided that it allows attaching structured metadata [26] (known as annotations or attributes) to program elements. This allows to use numerous existing tools and lowers barrier of learning new technologies for industrial programmers.

4. Case Study

The approach is demonstrated on the development of a new textual notation for the GraphML language. Graph Markup Language (GraphML) is a format for storage and exchange of graphs and associated metadata used by some graph drawing tools [5].

The translator was implemented using two tools: JAXB and YAJCo. JAXB is a standard solution for marshalling and unmarshalling Java objects to XML. YAJCo[3] (Yet Another Java Compiler Compiler) is a parser generator for Java that allows to specify language syntax using a metamodel in a form of annotated Java classes [30]. This allows declarative specification of the language and its mapping to Java objects [20]. In addition to the parser, YAJCo is able to generate a pretty-printer and other tools from the same specification [21].

This section describes the process of development of the translator using the chosen tools. It also explains challenges that arise during the implementation and their solutions. Readers can use it as a guide to develop their own translator[4].

As an additional illustration of the custom notation for an existing XML-based language, Appendix A provides example from our previous paper [6] — GtkBuilder language used to define graphical user interfaces.

[3] Available at https://github.com/kpi-tuke/yajco

[4] Complete source code of the translator is available at http://hron.fei.tuke.sk/~chodarev/graphl/

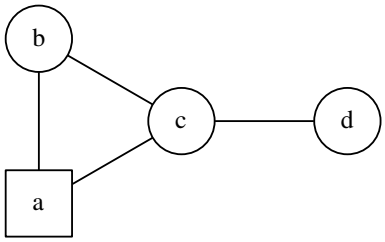

Fig. 2. Example graph

Listing 1. Example graph definition using XML notation

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <graphml xmlns="http://graphml.graphdrawing.org/xmlns">
3    <key id="ds" for="node" attr.name="shape" attr.type="string">
4      <default>circle</default>
5    </key>
6    <graph id="G" edgedefault="undirected">
7      <node id="a">
8        <data key="ds">square</data>
9      </node>
10     <node id="b"/>
11     <node id="c"/>
12     <node id="d"/>
13     <edge source="a" target="b"/>
14     <edge source="a" target="c"/>
15     <edge source="b" target="c"/>
16     <edge source="c" target="d"/>
17   </graph>
18 </graphml>
```

4.1. Graph Markup Language

A GraphML document contains one or more definitions of graphs and each graph consists from nodes and edges. In addition, nodes and edges can have user defined attributes attached. All possible attributes must be declared in the beginning of the document using a *key* element. GraphML also supports advanced concepts, like hyperedges (edges connecting more then two nodes), nodes with named ports (locations for edges to connect) and nested graphs.

An example definition for simple graph (depicted in Fig. 2) is presented in Listing 1. In addition to nodes and edges it defines one custom attribute with the name “shape” and default value “circle” (lines 3–5). The attribute is used to change shape of the node *a* to “square” (line 8).

The same definition in custom textual notation is presented in Listing 2. The notation is inspired by the DOT language [11]. Notation is much cleaner as it uses plain identifiers to define nodes and pseudo-graphical symbol “−−” to define edges. Additional properties of nodes or edges are specified in square brackets and may include port definitions or

Listing 2. Example graph definition using custom textual notation

```
key shape [id: ds, for: NODE, type: STRING, default: "circle"];
undirected graph G {
    a [ shape = "square" ] ;
    b ;
    c ;
    d ;
    a -- b ;
    a -- c ;
    b -- c ;
    c -- d ; }
```

attribute values. In contrast to XML notation, attributes are referenced by their name instead of ID, because custom name resolution strategy based on attribute names is used instead of default XML ID mechanism.

4.2. Project Layout

We recommend to split the project into two separate modules or subprojects:

1. the metamodel definition and the code generated based on it (parsers, pretty-printers),
2. the language translator that uses the metamodel and generated code from the first module.

This layout explicitly divides generated code and the code that depends on it, therefore simplifying build process. We used Apache Maven[5] to manage building and testing of the project and to configure multi-module project.

The first module contains only the classes representing metamodel and definitions of two concrete syntaxes. In our case, the concrete syntax definitions are attached directly to the metamodel classes in the form of Java annotations.

The second module contains implementation of the translator between the notations of the language. It instantiates JAXB marshaller and unmarshaller and also YAJCo generated parser and pretty-printer and uses them to read internal model of a document from one notation and write it in the other notation. This module also contains tests for automatic verifying of the translator and its parts (see section 4.6).

4.3. Metamodel Extraction

As was mentioned earlier, the metamodel represented by Java classes was generated based on the existing XML Schema using the XML binding compiler (`xjc`) that is a part of the JAXB. It generates Java classes corresponding to elements of XML-based language. Generated classes contain annotations that define mapping of classes and their fields to XML elements and attributes. JAXB uses these annotations to create instances of the

[5] Available at https://maven.apache.org/

Listing 3. NodeType class constructor with YAJCo annotations

```
@After(";") @NewLine
public NodeType(String id,
                @Before("[") @After("]") @Separator(",")
                List<NodeElement> dataOrPort) {
    this.id = id;
    this.dataOrPort = dataOrPort;
}
```

Listing 4. EBNF grammar rule generated from the NodeType class constructor

```
NodeType ::= <ID> <[> (NodeElement (<,> NodeElement)*)? <]> <;>
```

classes and set their properties based on XML document contents. The same annotations are used to serialize objects to the XML form.

This means that after the metamodel was extracted it is possible to use JAXB to read an existing graph definition from the XML notation to an internal representation defined by the metamodel and also to marshall the internal model back to the XML form.

In the case study, extracted classes directly corresponded to elements of the XML-based language. Therefore, they included classes like *GraphType*, *NodeType*, *EdgeType*, *KeyType* (declaration of data attribute), *DataType* (value of data attribute), etc. In total, 13 classes and 7 enum types was generated by JAXB.

4.4. Custom Syntax Definition

Definition of the new concrete syntax is also provided in the form of annotations added to the metamodel classes. This means that the metamodel generated using JAXB needs to be modified to include YAJCo-specific annotations.

While JAXB annotations are placed at classes and fields, in YAJCo most of the annotations are attached to constructors and their parameters. Each constructor is transformed into a grammar rule and parameters of the constructor determine the right-hand side of the rule. This allows to define syntactic alternatives for the same language concept and also explicitly specifying order of elements based on the order of parameters. In addition, YAJCo infers relations between language concepts from the inheritance relations between the metamodel classes.

For example, Listing 3 presents one of the constructors of the *NodeType* class. It defines that a graph node can be constructed from a string representing its identifier and a list of ports or data attributes (e.g. line 3 in Listing 2). The node definition would start with the *ID* token representing the identifier, followed by a sequence of elements enclosed in brackets and separated by comma. Grammar rule generated based on the constructor is presented in Listing 4. Annotations also contain hints on indentation and new-line placement that are used by the pretty-printer, but ignored by the parser.

Each variation of the element concrete syntax requires its own constructor. For example, the *node* can be defined with additional elements specified, or without them (for

example lines 3 and 4 in Listing 2) and therefore it needs at least two constructors. In addition to the constructors, factory methods can be used as an annotation target. This makes it possible to define different syntaxes even if they have the same types of parameters in Java.

Each class also needs a non-parametrized constructor required by JAXB. This constructor must be marked using the YAJCo `@Exclude` annotation so it would be ignored by the YAJCo tool.

4.5. Development Process

After defining initial subset of the concrete syntax, YAJCo was used to generate parser and pretty-printer. They were used to implement the translator according to the schema described in Section 3.

The translator was used to convert example XML documents taken from the official GraphML documentation[6] to the new syntax. Translated samples of the documents were manually checked by developers of the translator to evaluate the syntax. This process was repeated after each change of the syntax definition, therefore providing very short development cycles for experimenting with different notations for implemented language features.

The new notation was developed incrementally. Support for language concepts was gradually added and different variants of concrete syntax was considered and immediately evaluated by the authors of the translator. For example, several different notations for node attributes and ports was considered, before the final one was chosen.

4.6. Testing Translator Completeness

Development of the translator requires automatic testing of its completeness. This is done by executing round-trip translation — convert an XML document to the custom notation and then convert it back to XML. After the translation, contents of the document should not change, except of formatting.

To realize the testing, usual unit tests were implemented for each tested document. Comparison of XML documents was handled using XMLUnit[7] library that allows to perform XML comparisons for the purpose of application testing.

The same documents that were used to test the notation, was also used to test completeness of the translator. In addition to comparing original and resulting XML documents, unit tests printed intermediate form in the custom notation to help with notation checking.

4.7. Completing the Metamodel

The metamodel extracted from XML Schema can be incomplete in several ways. First of all, the XML Schema itself may be incomplete — missing definition of some language concepts or properties. On the other hand, the extraction tool can leave out some language properties or express them in a form that parser generator cannot understand. Language can also intentionally leave specification of some elements to extensions.

[6] GraphML Primer, available at http://graphml.graphdrawing.org/primer/graphml-primer.html
[7] Available at http://www.xmlunit.org/

Listing 5. Alternative types of values as defined by JAXB

```
public class GraphType {
    @XmlElements({
        @XmlElement(name = "data", type = DataType.class),
        @XmlElement(name = "node", type = NodeType.class),
        @XmlElement(name = "edge", type = EdgeType.class),
        @XmlElement(name = "hyperedge", type=HyperedgeType.class)
    })
    protected List<Object> dataOrNodeOrEdge;
    ...
}
```

Incomplete XML schema. Incompleteness of the schema can be easily solved by manual modification of the schema itself or extracted metamodel. For example, the schema of GraphML does not specify references between elements using identifiers. In most cases resolution of references is not required in a translator as it just passes identifiers from one notation to another without change. In the case of GraphML data attributes, however, we needed to access referenced data key. Therefore, generated classes was modified to include JAXB annotations `@XmlID` and `@XmlIDREF`.

Incomplete translation of the schema. This type of incompleteness is again solved by manual modification of the metamodel classes.

This problem appeared in cases, where several alternative values of different types are expected in the same context. For example, *graph* definition contains a sequence of nodes, edges, hyperedges or data attributes. In object-oriented model this situation can be expressed by inheritance. JAXB, however, does not use this technique in generated metamodel classes. Instead, it uses *Object* type in the container and adds `@XMLElements` annotation to specify all possible concrete types that can be used as is shown in Listing 5.

On the other hand, YAJCo requires the use of inheritance or implementation relations in these situations. So a new marker interface was created and classes of all elements that can appear in specific context are marked to implement it. The container class is then modified to reference the marker interface instead of the *Object*. Result of these modifications is presented in Listing 6.

Extensible languages. While first two cases can be easily fixed by manual or semi-automatic modification of the metamodel, the last one represents more complex problem. If all used extensions are known, it is possible to incorporate them into the metamodel. The other possibility is to include generic element type in the metamodel, that would represent all elements that are not defined explicitly. Then it would be possible to define also some generic notation for them in the custom form of the language that would be equivalent to XML.

Listing 6. Alternative types of values defined using inheritance

```
public class GraphType {
    @XmlElements( ... )
    protected List<GraphElement> dataOrNodeOrEdge;
    ...
}

public interface GraphElement {}

public class NodeType implements GraphElement { ... }

public class EdgeType implements GraphElement { ... }

public class HyperedgeType implements GraphElement { ... }
```

4.8. Model Transformations

In some cases it is useful to slightly modify abstract syntax for the purpose of custom notation. This can be done by defining separate metamodel and then transforming models from one metamodel to another. In simple cases, however, a single metamodel can be extended to include notation-specific properties.

Notation specific properties. Representation of the metamodel using Java classes allows to implement simple model transformations using constructors. Constructors of the metamodel classes can transform their parameters before storing to object fields. It makes it possible to define helper classes with own syntax rules, but store parsed information in a form expected by XML marshaller.

For example, hyperedges in XML notation contain endpoints and data attributes in arbitrary order. In textual notation, however, we need to separate them, because we decided to represent them differently: endpoints separated by "`--`" symbol and data attributes enclosed in brackets and separated by comma. In the result, hyperedge definition may look like this: `hyperedge: a -- b -- c [color="red", width="2.0"];`

To implement it we need to introduce separate constructor parameters for endpoints and data attributes that are used by YAJCo to generate parser. In addition, pretty-printer requires getters corresponding to these parameters.

As you can see in Listing 7, these separate lists are not even stored in the object fields. Instead, they are combined into existing field *dataOrEntrypoint*. The lists in the getters are constructed by filtering corresponding elements from the combined list. This means that constructor itself and getters implement this simple transformation.

Transforming visitors. Another way to implement transformation of the model is to introduce separate transformation step between reading the model in one notation and writing in the other notation. In object-oriented languages like Java, the Visitor design pattern can be used for this purpose.

Listing 7. Separate lists for endpoints and data attributes of hyperedge

```
public class HyperedgeType implements GraphElement {
    @XmlElements({
        @XmlElement(name = "data", type = DataType.class),
        @XmlElement(name = "endpoint", type = EndpointType.class)
    })
    protected List<Object> dataOrEndpoint;
    ...

    @Before({"hyperedge", ":"}) @After(";") @NewLine
    public HyperedgeType(
            @Separator("--") @Range(minOccurs = 1)
            List<EndpointType> endpoint,
            @Before("[") @After("]") @Separator(",")
            List<DataType> data) {
        this.dataOrEndpoint = new ArrayList<>(endpoint);
        this.dataOrEndpoint.addAll(data);
    }

    public List<EndpointType> getEndpoint() {
        return filterByType(dataOrEndpoint, EndpointType.class);
    }

    public List<DataType> getData() {
        return filterByType(dataOrEndpoint, DataType.class);
    }
    ...
}
```

This technique is especially useful for cases where values of fields used by both notations need to be modified. In our case it is used to change representation of strings and identifiers between notations (see next section).

4.9. Different Representations of Identifiers

An important problem arises from different treatment of keywords and other tokens in different notations. XML uses special syntax for language elements (tags delimited by angle brackets) and therefore it can allow to use language keywords as identifiers inside XML attributes and text fragments. For example, a graph can be named simply "graph": `<graph id="graph">...</graph>`

On the other hand, if element names like "`graph`" or "`hyperedge`" become reserved keywords in the custom notation, they could not be used as identifiers anymore, because standard lexical analyzer would not be able to distinguish them. Therefore equivalent expression "`graph graph {...}`" could not be parsed.

An ideal solution for the problem would be the use of scannerless parser [15]. It does not have separate lexical analysis step and therefore can distinguish identifiers and key-

words based on parsing context. In a case where it is not possible due to technological constraints, the conflict can be resolved in several ways:

1. by selecting language keywords with some special symbols that are not allowed in identifiers (for example, "`%graph`" instead of "`graph`"),
2. by requiring special notation for user defined identifiers (for example, beginning with the dollar sign "`$`"),
3. by modifying only conflicting identifiers using model transformation or in pretty-printer (for example, by appending underscore "`graph_`").

Another problem is in the definition of characters that are allowed in an identifier. If they are different, then illegal characters need to be replaced or escaped. This problem needs to be solved not only for identifiers, but also for other types of tokens, like strings.

Automatic sanitization of identifiers can be done in model transformation step described in the previous section. An alternative solution is to combine transformation in class constructor with customizing generated pretty-printer. In case of YAJCo, the pretty-printer is based on the Visitor pattern, so it can be easily customized by overriding needed methods in a subclass.

5. Recommendations

Experience from the case study can be summarized in several recommendations for application of the presented translator development approach. We suppose that most of these recommendations are not limited to used technologies and are applicable for development of any translator between two notations of the same language.

1. Use such representation of the language metamodel, that can be mapped to both translated notations and allows to automatically generate parser and pretty-printer from these mappings.
2. Use separate modules for the metamodel with generated code on one side and translator that uses it on the other side. This allows to define explicit dependencies between generated and handwritten code.
3. If it is possible, extract initial definition of the metamodel from specification of existing language notation. If resulting metamodel is incomplete, it can be completed manually.
4. Use notation-specific properties of model concepts to represent structural differences between notations. Simple transformations should be inserted in the translation process to convert values between these properties.
5. Take care of different representations of identifiers, strings and other types of tokens in different notations. This may require replacement or escaping of tokens during the transformation.
6. Use round-trip transformation of existing documents to test completeness of the translator and suitability of the new notation. This allows to see documents in the new notation without manually writing them.

6. Related Work

Domain-specific languages. Domain-specific languages are successfully used in different areas, for example specification of static structure of database applications [7], development of kiosk applications [40], or development of some specific aspects of applications like user interface [29], logging mechanisms [39], or acceptance tests [35]. It was also shown that DSLs improve comprehension of programs compared to general-purpose languages [17]. Therefore tools and methods for development of DSLs are active research topics.

Development of domain-specific languages can be guided by numerous patterns as described by Mernik et al. [22]. Since the approach described in this paper does not deal with design of a completely new language, not all patterns are applicable there. From the implementation patterns, the *Preprocessor* pattern clearly applies to our work. A language processor developed using the presented method does not perform complete static analysis of the code, so it is actually a preprocessor from the new concrete syntax to the old one.

Karsai et al. [14] provide guidelines for design of domain-specific languages. These guidelines are independent from concrete development approach and tools, so the guidelines from the *Concrete Syntax* category are fully applicable to the design of concrete syntax using our approach.

Application of usability testing methods to evaluation of DSLs is described in the work of Barišic et al. [2]. Kurtev et al. [19] also demonstrate that even low budget studies with small number of participants (Discount Usability Evaluation method) can be successfully used for this purpose. All these methods can be utilized during the design of the custom notation for DSL.

DSL development methods. A complete approach for the systematic development of domain-specific languages was presented by Strembeck et al. [36]. They define a model-driven development process for DSL development. Similarly to our approach, they suggest starting the process with the definition of the language model. In our case the language model is not developed based on domain analysis, but is extracted from the existing language specification. Behavior of the language and its integration with target platform are not defined, because they are provided by the existing implementation. Definition of the concrete syntax, however, can be done according to the process described in their work.

Villanueva Del Pozo in her thesis [38] defined an agile model-driven method for involving end-users in DSL development. The method proposes several concrete mechanisms to involve users in design and testing of the language based on questionnaires and specification of usage scenarios. Our approach supports similar ways of user involvement. In addition, in our case it is possible to use existing samples of DSL documents instead of usage scenarios.

To conclude, our approach differs from general-purpose DSL development methods in a fact that it does not cover design and implementation of a complete new language, but only design of the new concrete syntax for an existing DSL and implementation of the translator. Therefore we focus on aspects that are specific to this task and use the fact, that existing language definition and existing documents can be used to aid design, implementation and testing of the syntax and translator.

Alternative solutions. An alternative to development of the custom notation is the use of different generic language instead of the XML. YAML (Yet Another Markup Language) is a popular choice, for example Shearer [32] used it to provide textual representation for ontologies. YAML was specially designed as a human-friendly notation for expressing data structures [4]. Its syntax is readable, but the use of generic language does not allow to use specialized short-hand notations tailored for a developed language. While the basic structure of our example language may be expressed similar to the custom notation, problems start in the details. For example, the custom syntax uses infix notation for graph edges, that is not supported by YAML.

Similar solution is the use of OMG HUTN (Human-Usable Textual Notation) which specifies generic textual notation for MOF (Meta-Object Facility) based metamodels [23], again without possibility to customize concrete syntax.

XML-based language can be also replaced with an internal DSL that is embedded in a general-purpose language. For example, Nosal' and Porubän showed patterns for mapping XML to source-code annotations in case of configuration languages [27]. This approach, however, is limited to the specific type of languages that express application configuration and its mapping to elements of source code.

Another possibility is to derive textual notation automatically based on the metamodel. This approach was implemented for languages defined using Meta-Object Facility (MOF) [12]. Automatic derivation, however, does not allow to fine-tune the notation for the needs of users.

To conclude, all solutions based on some generic or automatically derived concrete syntax greatly simplify development process at the cost of restricted customizability of the syntax. Therefore, in cases where these limitations are acceptable, these methods may be more appropriate compared to the approach described in this paper. However, in cases where custom syntax is desirable, our approach can improve design and development process.

Alternative technologies. The approach presented in this paper does not depend on concrete tools, therefore it is possible to implement it using alternative technologies.

Neubauer et al. had shown in their work [24], that it is possible to use Ecore from the Eclipse Modeling Framework (EMF) [34] for representing metamodels and Eclipse Xtext [8] for generating parser, pretty-printer (*serializer* in the Xtext terminology), and editing support based on the Eclipse integrated development environment. They developed a tool, called XMLText, that automates development of round-trip transformation from XML-based languages defined by XML Schema to textual notation. Their tool also generates syntax definition for the language. This definition can be used as a starting point for customization using the process described in section 2, so our contribution compared to their work is in defining a tool-neutral development approach.

Another real-life example of migrating UML and XML based modeling languages to textual and graphical languages using EMF and Xtext was presented by Eysholdt and Rupprecht [9]. They, however, did not use a single metamodel for different notations. Instead, they used model-to-model transformations to migrate models.

The main difference compared to technologies presented in this paper is the fact that EMF and Xtext use specialized language for defining metamodel (Ecore), while JAXB and YAJCo rely on Java for this purpose. This allows to lower the entry barrier by mini-

mizing the amount of new technologies needed to be learned. It also allows to implement model transformations in Java using the techniques well-known by industrial programmers. On the other hand, EMF promises independence on concrete programming language. Together with Xtext they also provide a more mature platform for the development of language processors and editing environments. The approach itself, hovewer, is fully applicable using these tools as well.

Different language notations. The approach presented in this paper can be modified for other types of notations. The approach can be used, for example, in development of alternative notations for ontologies instead of XML-based languages, similarly to some existing tools [37, 10].

The new notation is also not required to be textual. As was shown in the work of Bačíková et al. [3], it is possible to use the same metamodel definition to generate a graphical user interface. This interface would consist of forms allowing to edit language sentences as an alternative to writing the model in textual form.

7. Conclusion

Presented case study showed the applicability of the model-driven translator development approach and therefore possibility of iterative design of language notation together with its translator to the original notation. It also allowed to formulate several recommendations for practical use of the approach. Most of them are not specific to the tools used in the study and should be applicable to other tools as well.

An advantage of the model-driven approach compared to grammar-driven approaches is in the fact that it allows to define concrete syntax variants as simple mappings to the abstract syntax and therefore to freely experiment with the concrete syntax, without the need to reimplement the whole translator.

Common representation of the model shared by several existing tools also allows to use them to construct complete translator with little effort. In our case study, Java classes was used as such common representation, therefore our work also showed that object-oriented programming language with support for annotations provide adequate foundation for expressing metamodels. This allows light-weight model-driven software development, that lowers barrier for adoption by allowing to use tools and knowledge from object-oriented programming.

The use of a custom notation, of course, have several disadvantages compared to the standard and well-supported notation such as the XML. It disables possibility to use existing tools like editors, code browsers and so on. If such tools are needed, they need to be developed by authors of the new notation, although this process can be supported by language development tools such as Xtext. Overall, benefits of custom textual notation compared to XML should be considered for each language individually based on possible improvements of the readability and environment in which the language is used.

Development of the case study also exposed several deficiencies and potential improvements in the YAJCo tool. Therefore, the future work would be devoted to its improvement. For example, built-in support for different types of tokens would greatly simplify language implementation, and generation of supporting tools, like editor, would improve experience of language users.

Acknowledgments. This work was supported by projects KEGA 047TUKE-4/2016 “Integrating software processes into the teaching of programming” and FEI-2015-23 “Pattern based domain-specific language development”.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley, Boston, USA (2006)
2. Barišic, A., Amaral, V., Goulão, M., Barroca, B.: Evaluating the Usability of Domain-Specific Languages. In: Software Design and Development, pp. 2120–2141. IGI Global (2012)
3. Bačíková, M., Lakatoš, D., Nosáľ, M.: Automatized generating of GUIs for domain-specific languages. In: CEUR Workshop Proceedings. vol. 935, pp. 27–35 (2012)
4. Ben-Kiki, O., Evans, C., Ingerson, B.: YAML Ain’t Markup Language. Version 1.2. Tech. rep. (2009), http://yaml.org/
5. Brandes, U., Eiglsperger, M., Lerner, J., Pich, C.: Graph Markup Language (GraphML). In: Roberto Tamassia (ed.) Handbook of Graph Drawing and Visualization, pp. 517–541. CRC Press (2013)
6. Chodarev, S.: Development of human-friendly notation for XML-based languages. In: Federated Conference on Computer Science and Information Systems (FedCSIS). pp. 1565–1571. IEEE (2016)
7. Dejanović, I., Milosavljević, G., Perišić, B., Tumbas, M.: A domain-specific language for defining static structure of database applications. Computer Science and Information Systems (ComSIS) 7(3), 409–440 (2010)
8. Efftinge, S., Völter, M.: oAW xText: A framework for textual DSLs. In: Workshop on Modeling Symposium at Eclipse Summit. vol. 32, p. 118 (2006)
9. Eysholdt, M., Rupprecht, J.: Migrating a Large Modeling Environment from XML/UML to Xtext/GMF. In: ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA). pp. 97–104. ACM, New York, USA (2010)
10. Fonseca, J.M.S., Pereira, M.J.V., Henriques, P.R.: Converting Ontologies into DSLs. In: 3rd Symposium on Languages, Applications and Technologies. OpenAccess Series in Informatics (OASIcs), vol. 38, pp. 85–92. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014)
11. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software: Practice and Experience 30(11), 1203–1233 (sep 2000)
12. Gargantini, A., Riccobene, E., Scandurra, P.: Deriving a textual notation from a metamodel: an experience on bridging modelware and grammarware. Milestones, Models and Mappings for Model-Driven Architecture p. 33 (2006)
13. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1), 31 – 39 (2008)
14. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S.: Design Guidelines for Domain Specific Languages. In: 9th OOPSLA Workshop on Domain-Specific Modeling (DSM’ 09). p. 7. No. October (2009)
15. Kats, L.C., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition. ACM SIGPLAN Notices 45(10), 918 (oct 2010)
16. Kleppe, A.: A Language Description is More than a Metamodel. In: Fourth International Workshop on Software Language Engineering. Grenoble, France (2007), http://doc.utwente.nl/64546/
17. Kosar, T., Oliveira, N., Mernik, M., Pereira, V.J.M., Črepinšek, M., Da Cruz, D., Henriques, R.P.: Comparing general-purpose and domain-specific languages: An empirical study. Computer Science and Information Systems (ComSIS) 7(2), 247–264 (2010)

18. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. In: International Symposium on Distributed Objects and Applications, DOA 2002 (2002), http://doc.utwente.nl/55814/
19. Kurtev, S., Christensen, T.A., Thomsen, B.: Discount method for programming language evaluation. In: 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU). pp. 1–8. ACM Press, New York, New York, USA (2016)
20. Lakatoš, D., Porubän, J., Bačíková, M.: Declarative specification of references in DSLs. In: 2013 Federated Conference on Computer Science and Information Systems (FedCSIS). pp. 1527–1534. IEEE (2013)
21. Lakatoš, D., Porubän, J.: Generating tools from a computer language definition. In: International Scientific conference on Computer Science and Engineering (CSE 2010). pp. 76–83 (September 2010)
22. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (dec 2005)
23. Muller, P.A., Hassenforder, M.: HUTN as a Bridge between ModelWare and GrammarWare - An Experience Report. WISME Workshop, MODELS/UML pp. 1–10 (2005)
24. Neubauer, P., Bergmayr, A., Mayerhofer, T., Troya, J., Wimmer, M.: XMLText: from XML schema to Xtext. In: 2015 ACM SIGPLAN International Conference on Software Language Engineering. pp. 71–76. ACM (oct 2015)
25. Nielsen, J.: Iterative user-interface design. IEEE Computer 26(11), 32–41 (Nov 1993)
26. Nosáľ, M., Sulír, M., Juhár, J.: Source code annotations as formal languages. In: 2015 Federated Conference on Computer Science and Information Systems (FedCSIS). pp. 953–964 (Sept 2015)
27. Nosáľ, M., Porubän, J.: XML to annotations mapping definition with patterns. Computer Science and Information Systems (ComSIS) 11(4), 1455–1477 (2014)
28. Parr, T.: Humans should not have to grok XML (8 2001), http://www.ibm.com/developerworks/library/x-sbxml/index.html
29. Perisic, B., Milosavljevic, G., Dejanovic, I., Milosavljevic, B.: UML profile for specifying user interfaces of business applications. Computer Science and Information Systems (ComSIS) 8(2), 405–426 (2011)
30. Porubän, J., Forgáč, M., Sabo, M., Běhálek, M.: Annotation based parser generator. Computer Science and Information Systems (ComSIS) 7(2), 291–307 (2010)
31. Raymond, E.S.: The art of Unix programming. Addison-Wesley (2004), http://www.catb.org/esr/writings/taoup/
32. Shearer, R.: Structured ontology format. In: Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions (2007)
33. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)
34. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
35. Straszak, T., Śmiałek, M.: Model-driven acceptance test automation based on use cases. Computer Science and Information Systems (ComSIS) 12(2), 707–728 (2015)
36. Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. Software: Practice and Experience 39(15), 1253–1292 (oct 2009)
37. Čeh, I., Črepinšek, M., Kosar, T., Mernik, M.: Ontology driven development of domain-specific languages. Computer Science and Information Systems (ComSIS) 8(2), 317–342 (2011)
38. Villanueva Del Pozo, M.J.: An agile model-driven method for involving end-users in DSL development. Ph.D. thesis, Universitat Politècnica de València, Valencia (Spain) (jan 2016), https://riunet.upv.es/handle/10251/60156
39. Zawoad, S., Mernik, M., Hasan, R.: Towards building a forensics aware language for secure logging. Computer Science and Information Systems (ComSIS) 11(4), 1291–1314 (2014)

40. Živanov, Ž., Rakić, P., Hajduković, M.: Using code generation approach in developing kiosk applications. Computer Science and Information Systems (ComSIS) 5(1), 41–59 (2008)

A. GtkBuilder Language Example

This appendix provides an example of the concrete syntax of the GtkBuilder language from our previous case study. GtkBuilder is a part of the GTK+ GUI toolkit that allows to declaratively specify layout of a user interface using an XML-based language[8]. Details of the implementation of the translator can be found in the original paper [6].

The GtkBuilder UI definition language allows to specify a layout of widgets forming a user interface and their properties using an XML notation. Each instance of a widget is defined using an *object* element, which contains its type, identifier, properties, signal bindings, and child objects. Listing 8 presents an example UI definition in the XML notation.

Listing 8. Example of user interface definition using XML notation

```
<interface>
  <object class="GtkDialog" id="dialog1">
    <child internal-child="vbox">
      <object class="GtkVBox" id="vbox1">
        <property name="border-width">10</property>
        <child internal-child="action_area">
          <object class="GtkHButtonBox" id="hbuttonbox1">
            <property name="border-width">20</property>
            <child>
              <object class="GtkButton" id="save_button">
                <property name="label" translatable="yes">Save
                </property>
                <signal name="clicked"
                        handler="save_button_clicked"/>
              </object>
            </child>
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

The same definition can be expressed using a custom notation as shown in Listing 9. The notation uses special symbols to provide concise representation for language elements. For example, *object* is expressed using "`[ Class id ... ]`" notation (e.g. line 1), properties are written simply as pairs in a form "`name : value`" (e.g. line 4),

[8] Specified at https://developer.gnome.org/gtk3/stable/GtkBuilder.html

Listing 9. Example of user interface definition using custom textual notation

```
1  [ GtkDialog dialog1
2  %child vbox :
3     [ GtkVBox vbox1
4     border-width : 10
5     %child action_area :
6        [ GtkHButtonBox hbuttonbox1
7        border-width : 20
8        %child :
9           [ GtkButton save_button
10          label : _ Save
11          clicked -> save_button_clicked ]]]]
```

signal binding is expressed as "`signal_name -> handler`" (line 11), and strings that should be translated in localized versions of UI are marked with underscore (line 10). The notation is short and quite intuitive at the same time.

Sergej Chodarev is Assistant professor at the Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his PhD. in Computer Science in 2012. The main areas of his current research are design and implementation of domain specific languages, meta-programming and user interfaces.

Jaroslav Porubän is Associate professor and the Head of Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in Computer Science in 2000 and his PhD. in Computer Science in 2004. Since 2003 he is a member of the Department of Computers and Informatics at Technical University of Košice. Currently the main subject of his research is the computer language engineering concentrating on design and implementation of domain specific languages and computer language composition and evolution.

Received: January 16, 2017; Accepted: September 21, 2017.