Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1565–1571
DOI: 10.15439/2016F530
ACSIS, Vol. 8. ISSN 2300-5963

Development of Human-friendly Notation for XML-based Languages

Sergej Chodarev
Technical University of Košice, Department of Computers and Informatics, Letná 9, Košice, Slovakia
Email: sergej.chodarev@tuke.sk

***Abstract*—XML is a popular choice for development of domain-specific languages. In spite of its popularity, XML is a poor user interface and a lot of languages can be improved by introducing custom notation. This paper presents an approach for development of custom human-friendly notation for existing XML-based language together with a translator between the new notation and XML. This approach is based on explicit representation of language abstract syntax that can be decorated with mappings to both XML and the custom notation. The approach supports iterative design and development of the language concrete syntax, allowing its modification based on users feedback. Development process is demonstrated on a case study of language for definition of graphical user interface layout.**

I. 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 [1].

While a more appropriate syntax can be chosen for 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 [2].

Development of the translator requires implementation of parser and generator. Proper separation of these two components also involves some internal representation of the language that would be created by the parser and then traversed to generate the XML. Development of all these components may be very tedious, even using parser generators.

This paper presents an approach to development of the translator that simplifies the process and allows to evolve the new syntax iteratively. The main idea of the approach is in extracting definition of language structure into a format that can be easily augmented with the definition of a new notation. 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 [3], can be used to generate a parser for the new notation. Connecting the parser with the XML unmarshaller one would get a complete translator from a custom human-friendly notation to the original XML-based.

Main topics discussed in the paper and its contributions are the following:

- It explains 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 II).
- The approach allows to develop a round-trip translator based on a single specification of abstract syntax. This enables *iterative development* of the notation in contrast to classical approach where complete syntax should be defined upfront. The process of iterative notation development is described in Section III.
- The whole approach is demonstrated on a case study of a language for specifying layout and properties of graphical user interface components (Section IV). The case study shows possible challenges of the approach and can be used as a guide to develop similar translators.

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 meta-data [4] (known as annotations or attributes) to program elements. This allows to use numerous existing tools and also avoids the need for special purpose representations and related technologies.

II. Model-driven Development of Language Translator

Similarly to model-driven software development [5] it is possible to drive development of the language translator by the model of the language – metamodel[2]. The metamodel defines language concepts with their properties and relations to other

[1]Java Architecture for XML Binding, available at https://jaxb.java.net/.

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

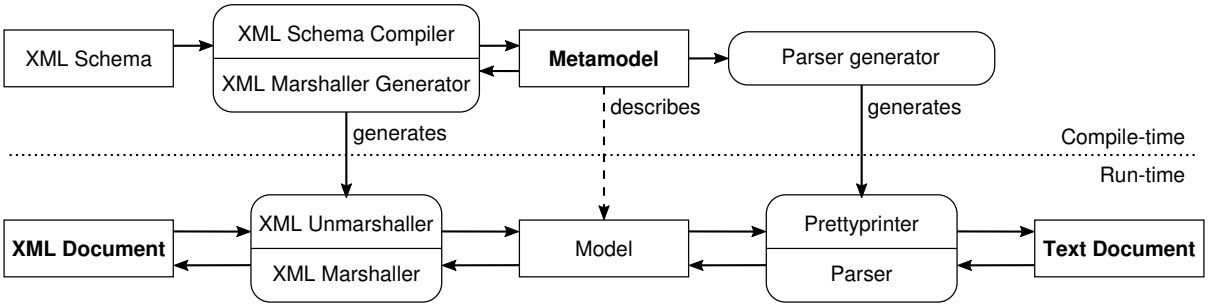

Fig. 1. Model-driven language translator development (arrows represent data-flow)

concepts. It can be annotated with additional information about concrete syntaxes of the language that need to be translated.

Figure 1 shows the whole architecture of 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 prettyprinter 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 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 – is derived automatically. This style of development also follows the "Single Point of Truth" principle, because the structure of the language is defined only once and its mappings to concrete notations are attached to it.

The described approach does not depend on concrete tools. It, however, requires an XML marshaller and a parser/prettyprinter generator that both use the same format for metamodel specification. In Section IV is presented the case study that uses Java classes 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 Eclipse Modeling Framework (EMF) [6] to represent the metamodel and Xtext [7] as a parser generator.

III. Iterative Development of the Translator

Design of notation for an existing language is, actually, design of a user interface. As such, it requires evaluation of various alternatives, and testing new alternatives in conditions similar to real-life. This process is iterative by nature [8]. On the other hand, classical approach to language development assumes that the language syntax is designed upfront (for example [9]). A complete specification of grammar is then augmented with semantic actions and processed to generate a parser. Changes in the syntax often require modification of semantic rules, making the process laborious.

The fact, that the model-driven approach described above allows to easily receive bidirectional translator, makes it possible to use a different process:

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

This process allows to easily use existing documents for testing 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.

This approach also provides a simple method for testing correctness of the developed translator, i.e. that no information is lost or corrupted during 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).

IV. Case Study

The approach can be demonstrated on the development of a new textual notation for the GtkBuilder language. GtkBuilder is a part of the GTK+ GUI toolkit that allows to declaratively specify layout of a user interface using an XML-based language[3]. There is a Glade tool[4] that allows to edit GtkBuilder specifications visually, however it tends to lack support for

[3]Specified at https://developer.gnome.org/gtk3/stable/GtkBuilder.html
[4]Available at https://glade.gnome.org/

```
1  <interface>
2    <object class="GtkDialog" id="dialog1">
3      <child internal-child="vbox">
4        <object class="GtkVBox" id="vbox1">
5          <property name="border-width">10</property>
6          <child internal-child="action_area">
7            <object class="GtkHButtonBox" id="hbuttonbox1">
8              <property name="border-width">20</property>
9              <child>
10               <object class="GtkButton" id="save_button">
11                 <property name="label" translatable="yes">Save</property>
12                 <signal name="clicked" handler="save_button_clicked"/>
13               </object>
14             </child>
15           </object>
16         </child>
17       </object>
18     </child>
19   </object>
20 </interface>
```

Fig. 2. Example of user interface definition using XML notation

newest GTK+ widgets, requiring manual modification of XML files.

The translator was implemented using two tools: JAXB and YAJCo. JAXB is a standard solution for marshalling and unmarshalling Java objects to XML. YAJCo[5] (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 [3]. This allows declarative specification of a language and its mapping to Java objects [10]. In addition to parser, YAJCo is able to generate pretty-printer and other tools from the same specification [11].

This section describes a 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 translators. The complete source code of the translator is available for download at http://hron.fei.tuke.sk/~chodarev/gtkbuilder/.

A. GtkBuilder Language

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. Fig. 2 presents an example UI definition in the XML notation.

The XML notation for the language, while familiar, is very hard to read. Document contains a lot of syntactic noise that makes fast scanning of the definition very hard. The same definition can be expressed using a custom notation as shown in Fig. 3. 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

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

```
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 ]]]]
```

Fig. 3. Example of user interface definition using custom textual notation

"`name : value`" (e.g. line 4), 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.

In the rest of the section the development of the custom notation and conversion tools is described in more detail.

B. Metamodel Extraction

As was mentioned earlier, the metamodel represented by Java classes can be generated based on the existing XML schema using the XML binding compiler (`xjc`) that is a part of JAXB. It generates Java classes corresponding to elements of XML-based language. Generated classes contain annotations that define mapping to XML elements and attributes. JAXB uses these annotations to create instances of the 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 existing UI definition from the XML notation to an internal representation defined by the

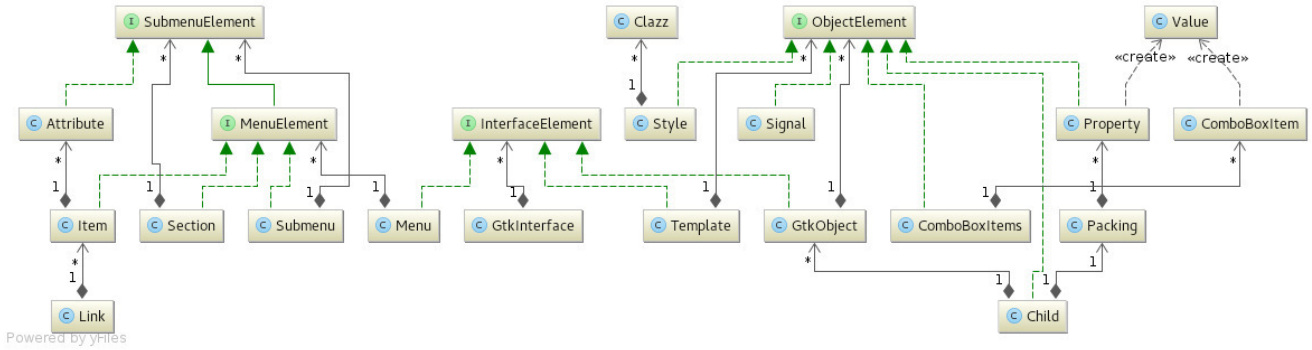

Fig. 4. Class diagram of the GtkBuilder language metamodel

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 *Interface*, *Object*, *Child*, *Property*, *Signal*, classes for definition of menus, etc. In total, 13 classes was generated by JAXB. Full metamodel, including modifications and additions described in next sections is depicted in Fig. 4.[6]

Encountered problems: Unfortunately, the schema of the GtkBuilder language is available only in the RelaxNG format. Because the support for RelaxNG schemas in JAXB is only experimental, it was converted to the XML Schema format using the Trang tool[7].

In addition, the schema does not define the language completely. Each widget type can support additional elements for widget-specific functionality. These elements, however, are not specified in the schema. Instead, arbitrary elements are allowed inside the *object* element.

The support for the most common widget-specific extensions, that was not specified in the schema, was added later by defining new classes in the metamodel. Generated classes obviously need to be modified to include declaration of added child elements of new types.

Shortcomings of the GtkBuilder language definition make it impossible to create the metamodel fully automatically. But on the other hand, it shows that the approach is applicable even in such cases.

C. Syntax Definition

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

YAJCo infers abstract syntax of the language from the inheritance relations between the metamodel classes and from their constructors. Each constructor is transformed into a grammar rule and parameters of a constructor determine the

[6]Classes *Object* and *Interface* was renamed to *GtkObject* and *GtkInterface* to avoid clashes with Java keywords in the generated parser.

[7]Available at http://www.thaiopensource.com/relaxng/trang.html

```
@Before("%child")
public Child(@Token("ID") @After(":")
             String internalChild,
             @NewLine @Indent
             List<GtkObject> object) {
    this.object = object;
    this.internalChild = internalChild;
}
```

Fig. 5. Example class constructor with YAJCo annotations

right hand side of the rule. YAJCo annotations are attached to constructors to specify details of the grammar that cannot be inferred automatically. For example, Fig. 5 presents one of the constructors of the *Child* class. It defines that a child can be constructed from a string representing an internal child name and a list of objects (e.g. lines 2 and 5 in Fig. 3). The child definition would start with the "`%child`" token followed by the `ID` token representing an identifier, followed by colon and a sequence of objects. Annotations also contain hints on indentation and new-line placement for prettyprinter (they are ignored by the parser).

Such constructors need to be added to the metamodel classes. Each variation of the element concrete syntax requires its own constructor. For example, the *Child* can be defined with *internalChild* property specified, or without it (e.g. line 8 in Fig. 3) and therefore it needs at least two constructors. In addition to 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.

In the following subsections are described some details of the implementation, typical problems and their solutions.

1) Completing the abstract syntax specification: In some cases several alternative values of different types are expected in the same place. For example, *object* definition contains a sequence of properties, child definitions or signal bindings. In

```
public class GtkObject {
    @XmlElements({
        @XmlElement(name = "property",
                    type = Property.class),
        @XmlElement(name = "signal",
                    type = Signal.class),
        @XmlElement(name = "child",
                    type = Child.class)
    })
    protected List<java.lang.Object>
        propertyOrSignalOrChild;
    ...
```

Fig. 6. Alternative types of values by as defined by JAXB

```
public class GtkObject {
    @XmlElements( ... )
    protected List<ObjectElement>
        propertyOrSignalOrChild;
    ...

public interface ObjectElement {}

public class Property implements ObjectElement{
    ...
public class Signal implements ObjectElement {
    ...
public class Child implements ObjectElement {
    ...
```

Fig. 7. Alternative types of values defined using inheritance

object-oriented model this situation can be expressed by inheritance. JAXB, however, does not use this technique in generated metamodel classes. Instead, it uses type *java.lang.Object* in the container and adds `@XMLElements` annotation to specify all possible concrete types that can be used as is shown in Fig. 6.

On the other hand, YAJCo requires the use of inheritance or implementation relations in these situations. So a new marker interface needs to be 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. An example of all these modifications is presented in Fig. 7.

2) Conflict between reserved keywords and identifiers: The problem arises from the different treatment of keywords 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, menu can be named simply "menu": `<menu id="menu">...</menu>`

On the other hand, if element names, like "`menu`" or "`child`", 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. Such conflicts can be resolved by decorating either language keywords or identifiers with some special symbols that would distinguish them. In our case percent sign was used as a starting symbol of all language keywords. For example, the menu would be defined like this: `%menu menu { ... }`

3) Model transformations: 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 class fields. It makes possible to define helper classes with own syntax rules, that are not stored in the model.

For example, at different places in the language it is possible to specify value, that can be a number, a symbol, or a string, all with different notations. Each class where such value can be used would need at least three constructors (for each notation of the value). Instead of this, it is possible to define a new helper class that would handle this aspect of concrete syntax using its own constructors and factory methods. It would also implement appropriate pre-processing of the values (e.g. removing quotation marks from strings). Instances of the helper class would become constructor parameters of the original classes, but only the actual value would be stored in the model, not the instance of the helper class. This allows to avoid modification of the metamodel and XML bindings.

D. Other Implementation Notes

1) Project setup: It is useful to split the project into two submodules: one for the metamodel definition and the code generated based on it, and other for code that uses the generated parser and prettyprinter to translate language sentences. This setup explicitly divides generated code and the code that depends on it.

The second submodule contains implementation of a command-line tool for converting between different notations of the language. This tool instantiates JAXB marshaller and unmarshaller and also YAJCo generated parser and prettyprinter and uses them to produce the internal model from one notation and convert it to the other notation.

In addition, the project contains script that tests the implementation by running round-trip transformation and comparing results with original versions of example documents. In the case study this script was implemented as a *Makefile* that would produce report on differences between documents if modifications of the code would cause errors in translation. This approach helped to find several problems described in this section and led to successful translation of tested examples.

2) Prettyprinter customization: Some syntax constructs can not be handled by the YAJCo generated prettyprinter automatically. For example, strings that should be translated in localized versions of the application are marked using an underscore "_" symbol. In the model, however, it is stored as a value of *True* in the field *translatable*. This correspondence is not inferred by the prettyprinter generator. As a solution, a prettyprinter can be simply extended by a new class, that would override the corresponding method to provide the needed functionality. This is greatly simplified by the fact that the generated prettyprinter is based on the visitor pattern.

V. Related Work

The presented approach and technologies are not limited to development of textual notation for the language. As was shown in the work of Bačíková et al. [12], 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. Input for the metamodel extraction is not limited to XML Schema: it is possible to extract metamodel from some non-XML notation [13], existing application [14] or it user interface [15]. It is also possible to avoid modification of generated metamodel code to augment definition of the metamodel and add different methods of its processing by using aspect-oriented programming [16].

The most similar work to the one presented in this paper is XMLText by Neubauer et al. [17]. They use EMF for representing metamodels and Eclipse Xtext [7] for generating parser, prettyprinter (*serializer* in the Xtext terminology), and editing support for the Eclipse integrated development environment. They integrate these tools and develop round-trip transformation between XML based languages defined by XML Schema and textual notation. Their tool, however, does not directly support custom syntax definition for each language element. On the other hand, customization of the textual notation should be possible using manual modification of the generated Xtext grammar.

Therefore, it should be possible to use the iterative approach described in this paper with EMF and Xtext as well. 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 minimizing 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 the concrete programming language. Together with Xtext they also provide a more mature platform for development of languages with their tooling, first of all – editing environment.

A real-life example of migrating UML and XML based modeling language to these technologies was presented by Eysholdt and Rupprecht [18]. They, however, did not use a single metamodel for different notations. Instead, they used model-to-model transformations to migrate models.

Other alternative would be the use of different generic language instead of the XML. YAML is a popular choice, for example, Shearer [19] used it to provide textual representation for ontologies. YAML (Yet Another Markup Language) was specially designed as a human-friendly notation for expressing data structures [20]. Its syntax is readable and quite simple, 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 notation allows to mark any string as translatable by simply writing underscore before it, YAML would require a different and more noisy solution.

Similar solution is the use of OMG HUTN (Human-Usable Textual Notation) which specifies generic textual notation for MOF (Meta-Object Facility) based metamodels [21], again without possibility to customize concrete syntax.

The approach presented in this paper is also similar to tools supporting development of DSLs based on existing ontologies [22], [23]. In our case, however, existing XML-based language is used as a basis for a DSL instead of ontology.

VI. Conclusion

Presented case study showed the applicability of the model-driven translator development approach. It also allowed to formulate several advises for practical usage of the approach (described in Section IV). While most of them are specific to the tools used in the study, some may be applicable to other tools as well. The approach itself is tool-independent and can be used with any language metamodel representation that can be mapped to both XML and custom textual syntax.

Future work may include identification, validation and comparison of tools and metamodel representations that support the described translator development approach. The YAJCo tool itself requires further development, especially in the area of generating tool support for the language beside parser and prettyprinter.

Acknowledgment

This work was supported by project KEGA No. 047TUKE-4/2016 “Integrating software processes into the teaching of programming”.

References

[1] T. Parr, “Humans should not have to grok XML,” 8 2001. [Online]. Available: http://www.ibm.com/developerworks/library/x-sbxml/index.html

[2] 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

[3] 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

[4] M. Nosáľ, M. Sulír, and J. Juhár, “Source code annotations as formal languages,” in *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sept 2015. doi: 10.15439/2015F173 pp. 953–964.

[5] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN 0470025700

[6] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[7] S. Efftinge and M. Völter, “oAW xText: A framework for textual DSLs,” in *Workshop on Modeling Symposium at Eclipse Summit*, vol. 32, 2006, p. 118.

[8] J. Nielsen, “Iterative user-interface design,” *IEEE Computer*, vol. 26, no. 11, pp. 32–41, Nov 1993. doi: 10.1109/2.241424

[9] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811

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

[11] D. Lakatoš and J. Porubän, "Generating tools from a computer language definition," in *Proceedings of International Scientific conference on Computer Science and Engineering (CSE 2010)*, September 2010, pp. 76–83.

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

[13] J. Porubän, J. Kollár, and M. Sabo, "Abstraction of computer language patterns: The inference of textual notation for a dsl," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 2012, pp. 365–385, doi: 10.4018/978-1-4666-2092-6.ch013.

[14] J. Kollár and M. Vagač, "Aspect-oriented approach to metamodel abstraction," *Computing and Informatics*, vol. 31, no. 5, pp. 983–1002, 2012.

[15] M. Bačíková, J. Porubän, S. Chodarev, and M. Nosáľ, "Bootstrapping DSLs from user interfaces," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC '15*. ACM Press, apr 2015. doi: 10.1145/2695664.2695994. ISBN 9781450331968 pp. 2115–2118.

[16] J. Porubän, M. Sabo, J. Kollár, and M. Mernik, "Abstract syntax driven language development: Defining language semantics through aspects," in *Proceedings of the International Workshop on Formalization of Modeling Languages (FML '10)*. New York, NY, USA: ACM, 2010. doi: 10.1145/1943397.1943399. ISBN 978-1-4503-0532-7

[17] P. Neubauer, A. Bergmayr, T. Mayerhofer, J. Troya, and M. Wimmer, "XMLText: from XML schema to Xtext," in *2015 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, oct 2015. doi: 10.1145/2814251.2814267. ISBN 978-1-4503-3686-4 pp. 71–76.

[18] M. Eysholdt and J. Rupprecht, "Migrating a large modeling environment from xml/uml to xtext/gmf," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010. doi: 10.1145/1869542.1869559. ISBN 978-1-4503-0240-1 pp. 97–104.

[19] R. Shearer, "Structured ontology format," in *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions*, 2007.

[20] O. Ben-Kiki, C. Evans, and B. Ingerson, "YAML Ain't Markup Language. Version 1.2," Tech. Rep., 2009. [Online]. Available: http://yaml.org/

[21] P.-A. Muller and M. Hassenforder, "HUTN as a Bridge between ModelWare and GrammarWare - An Experience Report," *WISME Workshop, MODELS/UML*, pp. 1–10, 2005.

[22] I. Čeh, M. Črepinšek, T. Kosar, and M. Mernik, "Ontology driven development of domain-specific languages," *Computer Science and Information Systems (ComSIS)*, vol. 8, no. 2, pp. 317–342, 2011. doi: 10.2298/CSIS101231019C

[23] J. M. S. Fonseca, M. J. V. Pereira, and P. R. Henriques, "Converting Ontologies into DSLs," in *3rd Symposium on Languages, Applications and Technologies*, ser. OpenAccess Series in Informatics (OASIcs), vol. 38. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014. doi: http://dx.doi.org/10.4230/OASIcs.SLATE.2014.85. ISBN 978-3-939897-68-2. ISSN 2190-6807 pp. 85–92.