Automatized Generating of GUIs for Domain-Specific Languages

Michaela Bačíková, Dominik Lakatoš, and Milan Nosáľ

Technical University of Košice, Letná 9, 04200 Košice, Slovakia,
(michaela.bacikova, dominik.lakatos, milan.nosal)@tuke.sk,

Abstract. Domain-specific languages (DSLs) promise many advantages over general purpose languages (GPLs) and their usage is on the rise. That is one of the reasons for us at our university to teach the subject called Modelling and Generating of Software Architectures (MaGSA), where the students learn how to work with DSLs and their parsers. Our students often have problems understanding a new DSL, even if it is very simple, because they never have worked with a DSL before. Therefore we decided to find a way to help them with the learning process by providing a DSL-specific editor. This editor will be automatically generated based on a language specification and the generator will be integrated into the existing infrastructure of the MaGSA project, which the students use on the subject. In this paper we present various techniques to reach the goal of generating graphical editors for DSLs and we introduce a prototype which is able to automatically generate a user interface for an external DSL.

Keywords: domain-specific languages, graphical user interface, generative programming, metamodelling

1 Introduction

DSLs [1] promise advantages over GPLs. As mentioned by Brooks [2], programmers tend to produce statements at the same rate whatever language they use, so high-level languages are more productive. Still, only a few programmers go to the trouble of creating a new language as it comes with a plenty of problems lengthening the development process. It is hard to get it right at the first time without the proper knowledge: it is a new syntax to learn and maintenance can be expensive [3]. This usually leads to systems written in GPLs which, however, do not support domain-specific abstractions very well. A programmer ordinarily ends up with a code containing a lot of computer-specific programming noise and a lot less of the written code is really useful for solutions of domain problems.

At our university we teach a subject called Modelling and Generating of Software Architectures (MaGSA) which tries to show our students how to use model-based [4, 5] and generative programming [6, 7] for creating software systems. On this subjects' exercises the students use a pre-prepared project and tutorials, all constructed in a way enabling them to understand both the complexity of developing a rather simple language parser with several application generators and the simplicity of using a DSL to configure the desired application. It shows the possibilities of such an application generator tool to create different software applications by just changing the input file and running the implemented parsers and generators. We call the DSL used on the exercises the *entity language*. Metamodel of this DSL is defined by Java classes and its grammar is defined by an internal DSL (i.e. annotations) in the project and the *YAJCo (Yet Another Java Compiler Compiler)* parser generator generates a language processor for this DSL. YAJCo tool and it's advantages were completely described by Porubän [8, 9].

In our exercises we deal with problems related to incorrect understanding of the subjects' curriculum by our students. The primary problem is that the students do not get the picture of the syntax and the semantics of the entity language defined in our project i.e. they use it incorrectly and it takes some time for them to fully understand it and learn to work with it. Since no specific editor for the entity language was created yet, the students work with free text editors such as Notepad or Notepad++ which do not provide any syntax highlighting for our DSL and it has no code completion. So they do not give the students any hint for writing the code in the entity language. Therefore an idea arises to create a specific editor for the entity language, to be able to help the students learning a new language and to write code in it easily.

There is however still one important feature, namely the possibility of changing the DSL specification by changing the classes and annotations in the project - if it is because of changes in the subject curriculum or simply because of language evolution [10]. If the generator should be generally usable we can not give the students a one-off tool for a single DSL, which can not be changed. Moreover, the entity language specification really changes during the development process on the exercises. Therefore we decided to perform an experiment of generating a specific editor for the entity language. The main idea of the solution is similar to the idea of language workbenches described by Fowler [1, 11]. However the main difference in our approach is that the desired output is not a classical editor allowing the common integrated development environments like automatic code completion or syntax highlighting. It is rather a graphical interface with a support for creating new language constructions similar to CRUD (Create Retrieve Update Delete) interfaces.

There are many existing language workbenches that could help with graphical environment, code completion, syntax highlighting or refactoring features, for instance Eclipse Modeling Framework [14] and Eclipse Graphical Modeling Project [15], Spoofax [13], MetaEdit+ [16] or the Generic Modeling Environment [17]. The main reason for us to decide not to use them and go for the way of generating our own editor with a CRUD interface is *simplicity* and *understandability*. The students would learn to create new sentences in the language and they also would learn the relations that the user interface represents. The second reason is that these workbenches have integrated generators and language parsers, which are not compatible with the project used on our exercises and we want our students to create their own application generators which is the goal of the subject. The advantage is that once the editor generator is included in the MaGSA project, the students will have a chance to take a look into the techniques of generating simple editors for DSLs which they can further use in the subject. We present this paper as another solution to an existing problem of DSL learnability and we think that graphical user interfaces (GUIs) for DSLs represent a good first step for motivating students to use DSLs.

1.1 Tasks and goals

In this paper we try to introduce a first step on a way to the solution of automatic generating of GUIs for external DSLs. The possibility of derivating the user interface specific for a DSL based on its specification was already proved. We, however, try to present a different way of solving this problem which would shorten learning time of DSLs in the first phases of learning. Our general goal is to implement a prototype which will be able to generate graphical user interface based on a language specification. Based on different types of statements in the language specification it will be able to generate different basic graphical components. We will introduce various partial techniques to creating this solution and also different techniques to reach the general goal. The partial techniques will be illustrated by examples.

The goal of this paper is to design a method and to implement a prototype enabling automatic generating of a user interface for DSL based on language metamodel. The main goal is a definition of a solution, that will guide solvers to the right direction to the realization of a project. The most important property of our solution is simplicity and learnability.

2 The MaGSA Project

The prototype is based on an existing project, which is currently in the teaching process of the MaGSA subject on the Technical University of Koice, Faculty of Electrotechnical Engineering and Informatics for teaching metamodelling, DSLs and generative approach to software development. For the purpose of simplification, in the rest of the paper we will refer to this project as to the 'MaGSA project'. The MaGSA project used by our students is a pre-prepared project implemented in the Java language and it uses the standard MDA architecture. The exercises are realized in a form of a tutorial, which provide the description of the project and introductory source codes, with which the students work. We kindly ask the readers to check the official website in English [12] for further information about the MaGSA project and the tutorials.

Two basic entities are defined in the model of MaGSA project: Employee and Department. The model of the MaGSA project is written in the entity language and its metamodel can be described by a grammar as follows:

```
Model ::= ((Entity (Entity)*) (Reference)*)
Reference ::= (<reference> <NAME> <NAME>)
```

```
Entity ::= (<entity> <NAME> (<{> (Property (Property)*) <}>))
Property ::= (<NAME> (<:> Type) ((Constraint ((<,> Constraint))*))?)
Type ::= (<INTEGER> | <REAL> | <STRING>)

Constraint ::= (Regex | Required | Range | Length | Contains)
Regex ::= (<regex> <STRING\_VALUE>)
Required ::= <required>
Range ::= (<range> <INT_VALUE> <INT_VALUE>)
Length ::= (<length> <INT_VALUE> <INT_VALUE>)
Contains ::= (<contains> <STRING_VALUE>)
```

Terminals are noted in brackets ⟨ ⟩.

The basic entities of the metamodel and the relations between them are defined by the class implementation. The concrete syntax is defined by means of annotations in the individual classes. Based on these classes and annotations the YAJCo tool generates the grammar along with the language processor for the DSL.

2.1 Entity Language

As we mentioned earlier, the model of the generated application, is defined by an external DSL called the *entity language*, usage of which can be seen on an example file written in the entity language below.

```
# Department
entity Department {
    name : string required, length 5 30
    code : string required, length 1 4
    level : real
}

# Employee
entity Employee {
    name : string required, length 2 30
    surname : string required, length 2 30
    age : integer
}
reference Employee Department
```

On the MaGSA subject the students work with these two entities, but it is possible to define any number of entities and references between them and we encourage students to do so. As we mentioned earlier, they have problems grasping the basic idea, therefore only a very small amount of students do so without any help.

In the current state of the MaGSA project, the input file written in the entity language is processed by the generated language processor and based on the information processed the resulting application is generated, which provides a console interface to be able to work with the concrete departments and employees saved in the database. The task of the students is to create generators which generate the console application.

The students work with the entity language only manually which, when working with a big number of entities, could be tedious and exhausting. The problem can arise also when the metamodel is changed. In that case manual correction of each entity in the entity language file is needed. Therefore tool enabling working with the entity language would be helpful. The tool should enable adding new entities, their editing and removing. This tool would be generated based on the metamodels' grammar specifically for any defined DSL.

3 Problem Analysis

The challenge of our goal is the creation of a new generator of a DSL-specific GUI which would reflect the relations between the language concepts and thus support the domain-specific abstractions. The basic idea is best to explain on the grammar shown in the chapter 2.

The left sides of the rules define the points, which can be expressed by forms of the user interface. The right sides of the rules define the content of the forms.

In the EBNF we know these tree basic operators:

- sequence,
- alternative,
- repetition 0-* or 1-*,

and two basic types of symbols:

- terminal,
- non-terminal.

A sequence of symbols transforms into a GUI as a list of elements in a particular form. If the sequence contains terminal symbols, then these symbols have their type. Based on this type the corresponding component can be derived as illustrated in Tab. 1.

Table 1. Types of grammar symbols and components derived based on these types

Value type	*Derived component*
<STRING_VALUE>	text field
<INTEGER_VALUE>	jspinner or a number field (a text field with a number input)
<NAME>	selection with existing named entities

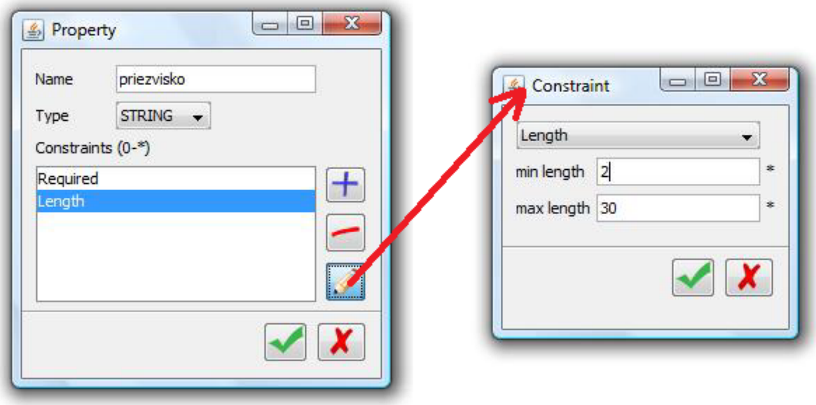
If a given symbol is a nonterminal then after clicking on a particular GUI element or an item a new dialog for editing this element or item opens.

A reduction of the number of dialogs can be made. If a rule contains only alternatives and no other elements or operators then it is possible to generate a dialog with a combo-box on the top and the remaining content will change dynamically based on what was selected in the combo-box. The content represents the particular alternative.

The following example shows two production rules for Property and Constraint.

```
Property ::= (<NAME> (<:> Type) ((Constraint ((<,> Constraint))*))?)
Constraint ::= (Regex | Required | Range | Length | Contains)
```

Fig. 1 illustrates a GUI for the `Constraint` and `Property` rules of our grammar. In the Property dialog new constraints can be added by clicking the '+' button. Consequently the dialog with one of the constraint types defined for the property can be chosen an its content automatically changes based on the particular alternative representation.



Name: priezvisko
Type: STRING
Constraints (0-*): Required, Length

Length
min length: 2
max length: 30

Fig. 1. Illustration of a GUI generated from the Property grammar rule

Based on each type of EBNF operator it is possible to derive different types of components. Tab. 2 lists the components that can be derived based on a particular EBNF operator.

We list different types of components for each operator that can be used for the implementation, because they meet the same purpose. Each of them can be chosen for the implementation, the only deciding factor is usability.

As can be seen on these examples, the components reflect the relations between the language concepts. It can be seen instantly that the Property has a name, a type and a list of constraints. We assume this can help grasping the idea of DSLs.

Table 2. The types of operators in the grammar and components derived based on these types

Operator	*Derived component*
Alternatives (the — operator)	Combo-box
+ (repeat 1-*)	A combo-box or a radio button (with single selection) + 3x CRUD button
* (repeat 0-*)	A list or check-box buttons (with multiple selection)

4 Prototype Implementation

A prototype was implemented in the Java language into the existing infrastructure of the MaGSA project. A new generator was created by extending the abstract generator provided by the project. The metamodel is defined by the metamodel classes and annotations for the YAJCo parser generator and a user interface is generated to enable the model defined by the entity language. It is possible to add new entities, edit existing entities and to remove entities. Also, in each entity a further change of all properties defined by the model is possible. It is possible to add references from one entity to other. The initial screen of the application is in Fig. 2. It contains a list of all defined entities and a list of references. It is possible to create new files in the entity language with the *.el extension, or open existing files.

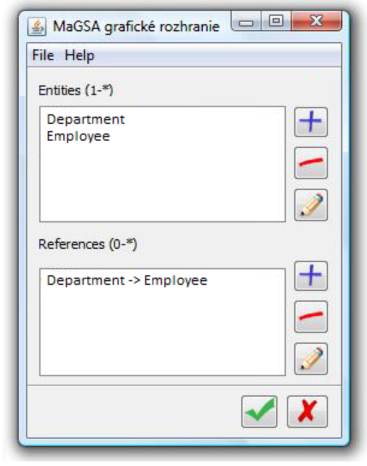


Fig. 2. Illustration of GUI generated from the grammar

We realize that this prototype is implemented in the specific environment of our MaGSA project, but the idea is usable also in other areas. We believe it will help beginners to learn the basic principles behind DSLs.

5 Conclusion

In this paper we introduced a first step on a way to the solution of automatic generating of GUIs for external DSLs. We showed a simple way of deriving a GUI for a DSL specification and of using the GUI to create new files written in this DSL. The DSL-specific interface is simple to use even for a student and it is more understandable than a classic textual editor. Reflecting relations between the domain-specific entities ensures the support of domain-specific abstractions. We introduced various partial techniques to creating this solution, illustrated by examples and also different techniques to reach the general goal. We created a prototype integrated into the existing infrastructure of the MaGSA project which is able to generate basic graphical elements based on the DSL specification. This way students will be guided to the right direction to the realization of the MaGSA project. In the future we plan to perform usability experiments with our students prototype to be able to improve the prototype further. The tool will be used by one group of students and another group will work without the tool. We will measure the learning time, completion time and failure rate. In the end we will evaluate subjective usability using questionnaires. We will compare the results between the two groups to determine whether the tool is really helpful.

6 Acknowledgement

This work was supported by VEGA Grant No. 1/0305/11 - Co-evolution of the artifacts written in DSLs driven by language evolution.

References

[1] Fowler, Martin: Domain-Specific Languages (Addison-Wesley Signature Series). Addison-Wesley Professional (2010)
[2] Brooks, F.P.: The Mythical Man Month: Essays on Software Engineering. Addison Wesley Professional Longman Aniversary Ed. (1995).
[3] Freeman, S., Pryce, N.: Evolving an Embedded Domain-Specific Language in Java. OOBSLA'06 October 22-26, Portland, Oregon, USA (2006).
[4] T. Stahl, M. Voelter: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006).
[5] J. Greenfield, K. Short, S. Cook, S. Kent: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004).
[6] Czarnecki, K. Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional (2000).
[7] Chen, J.: Generators. Presentation. USA, University of Colorado, march $http : //wiki.lassy.uni.lu/se2c - bib_download.php?id = 1117$ (accessible 27.5.2009) (2003).
[8] 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 Modelling Languages, New York, NY, USA, (2010).

[9] J. Porubän, M. Forgáč, M. Sabo and M. Běhálek: Annotation Based Parser Generator. Computer Science and Information Systems, vol. 7, no. 2, pp. 291-307, (2010).

[10] Bell, P.: DSL Evolution. Article. December http://www.infoq.com/articles/dsl-evolution (accessible 20.4.2011) (2009).

[11] Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html, (accessible 27.5.2009)(2005).

[12] Modelling and Generating of Software Architectures. Student tutorial, English version. Technical University of Koice, Faculty of Electrical Engineering and Informatics, Department of Computers and Informatics http://hornad.fei.tuke.sk/ bacikova/MaGSA (2010)

[13] The Spoofax Language Workbench http://strategoxt.org/Spoofax (accessible 05.08.2012)(last update 2012)

[14] Eclipse Modeling Framework http://www.eclipse.org/modeling/emf/ (accessible 05.08.2012) (projets' last update 2009)

[15] Graphical Modeling Project http://www.eclipse.org/modeling/gmp/ (accessible 05.08.2012) (projets' last update 2010)

[16] Domain-specific Modeling with MetaEdit+ http://www.metacase.com/ (accessible 05.08.2012) (projets' last update 2012)

[17] Generic Modeling Environment http://www.isis.vanderbilt.edu/Projects/gme/ (accessible 05.08.2012) (projets' last update 2008)