Baltic J. Modern Computing, Vol. 2 (2014), No. 2, 56-74

Configuration Language for Domain Specific Modeling Tools and Its Implementation

Arturs SPROGIS

Institute of Mathematics and Computer Science, University of Latvia,
Raina blvd. 29, LV-1459, Riga, Latvia

arturs.sprogis@lumii.lv

Abstract. The paper proposes an approach how to develop a configurator (tool) allowing defining a wide range of DSML tools. The configurator is based on the idea that DSML tool definition may be given by a universal metamodel (i.e., UML class diagram), and then it can be interpreted by a universal interpreter to obtain the working tool. But for non-standard cases when the existing metamodel facilities are limited, the extension point mechanism is introduced allowing adding a tool-specific functionality by specifically programming it. The developed configurator has been verified in several practical applications and has proven its effectiveness.

Keywords. Configurator, DSML, DSML tools, specification, metamodel, metamodeling

Introduction

Domain Specific Modeling Language (DSML) (Kelly and Tolvanen, 2008) and DSML tools have become increasingly more important in the context of system modeling (Sprinkle, 2007). We can use UML (WEB, l), BPMN (WEB, a), SysML (WEB, k), and the corresponding heavy weight tools, but in such case a number of problems arise. Firstly, heavy weight tools are built by IT people and they are well understood by IT people, but not by business people. Secondly, universal languages are designed to describe a wide area of domains, whereas in real applications only a few language constructs are used and unused constructs complicate the tool's usability. Thirdly, heavy weight tools have a fixed list of features, and there is no mechanism to add new domain specific features, for instance, model consistency checking, text generation from models, connections to external databases, etc. Finally, the internal structure of such tools is not open for external programs and therefore external programs cannot access the model data.

DSML tools solve most of these problems by developing a tool specific functionality for each application area (Cook et.al., 2007). But there is another problem, namely, it takes a lot of effort to program a single DSL tool from scratch (Robert et.al., 2009).

The way DSML tool development process can be made simpler and quicker is to develop a universal tool building platform. Since platform development is complicated and expensive, there are not many tool building platforms, and the most popular platforms are Eclipse EMF (WEB, b), Microsoft DSL (WEB, p) and MetaEdit (WEB, g).

Typically DSML tool building platforms exploit two approaches. The most common approach is used by Eclipse EMF and Microsoft DSL and it consists of three steps. The first step creates an abstract syntax model of DSML, the second step creates the concrete syntax model and the third step establishes mappings between the two models. The purpose of such architecture is to have an exactly one abstract syntax model, which can be mapped to an arbitrary number of concrete syntax models. Platforms exploiting this approach provide a fixed functionality, but if extra features are needed they have to be programmed additionally that is usually not easy.

This architecture has two benefits. Firstly, it is more convenient to process complex data in abstract syntax model instead of concrete syntax model. Secondly, this approach ensures automatic data synchronization between concrete syntax models, namely, the changes in one of concrete syntax models are first introduced in the abstract syntax model, and then automatically synchronized with other concrete syntax models.

Obviously, this approach has advantages, if there are many concrete syntax models or there is a necessity for complex data procession. However, if there are one or two concrete models and data processing is relatively simple as it is mainly in DSML tools, it is sufficient to have only concrete syntax models in the platform while abstract syntax model with its mapping establishment unnecessarily complicates the DSML tool development process.

The second approach is used by MetaEdit, and it states that DSML tools are specified only by concrete syntax model and afterwards this model is interpreted. In many cases this approach is very convenient and allows creating DSML tools quickly and easily. However, DSML world is complex, and it is not possible to cover all the use-cases with a single metamodel and the interpreter with a fixed functionality.

By comparing both approaches we may conclude that both methods have its limits, namely, the first approach allows defining a broad class of DSML tools, but it is difficult, whereas the second approach allows defining DSML tools in a relatively easy way, but it is limited to the functionality of the interpreter. Thus, the problem is to find the method that, firstly, allows developing DSML tools in an easy way, secondly, is not limited to a fixed functionality, but allows adding an additional functionality for non-standard cases by programming it and, thirdly, provides tool openness for accessing model data by external applications.

The paper presents an approach allowing defining a broad class of DSML tools as well as satisfies the mentioned three points.

1. The context

To implement the proposed approach, TDA tool building platform (Kozlovics, 2012), (Barzdins et.al., 2008), (Barzdins et.al., 2013) and model transformation language lQuery (Liepins, 2011), (Liepins, 2012) is used. In more detail we will examine the structure of TDA, and a schematic representation of TDA is shown in Fig. 1. The main components of TDA are engines and metamodels. Every engine provides a particular service in the tool, for instance, the presentation engine (Barzdins et.al., 2009b) visualizes diagrams and provide an interface between tool users and diagrams, but the dialog engine (Kozlovics, 2010) constructs dialog forms and provide an interface between tool users and dialog forms. Whereas metamodels are data schemas, that store the information of the tool state for engines.

Technically speaking, the platform is based on three components - a model repository, model transformations and an event-command mechanism. An event-command mechanism provides an interaction among tool users, engines and model transformations. In particular, the working tool is provided by an engine, and if a tool user performs an action, for instance, double-clicks on an element in the diagram, the user's action is "caught" and classified by the engine that creates a corresponding instance in the repository. Each engine has a fixed list of events and every event is attached to the model transformation that processes the event. Thus, when the engine has "caught" the user action and has created an instance of an event in the repository, then the control is transferred to the model transformation that is responsible for the procession of this type of event.

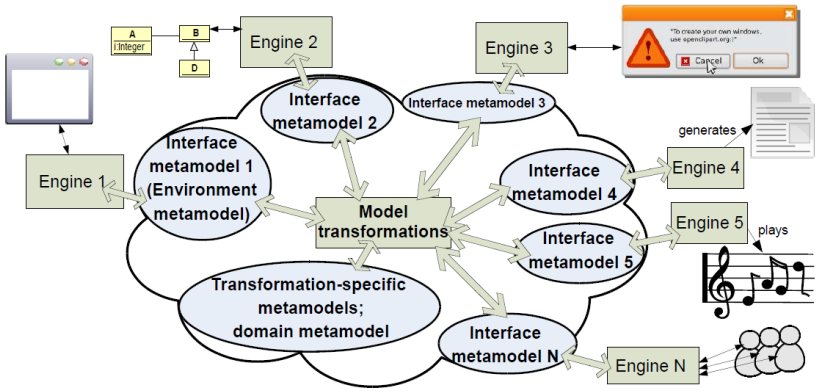

Fig. 1. Transformation-Driven Architecture (the author of the image is S. Kozlovics)

Transformations perform an event processing and for this reason they can be considered the "brains" of the platform. They are allowed to change the contents of the repository, for instance, to create new diagram elements or change element values. When transformations finish the execution of the assigned task, they return the control of the tool to the engine which assigned the task. In case the transformation must have to give an order to do something for the engine, for instance, to order to re-draw the diagram (for example, if a new element was created), the transformation has to create a new instance of the corresponding command in the repository. Every engine has a fixed list of supported commands and when an order is issued, the engine performs appropriate actions.

There are three engines (and their corresponding meta-models) implemented in the platform. The presentation engine visualizes diagrams stored in the repository and provides tool functionality - palettes, toolbars, etc. The dialogue engine constructs dialogue forms from instances of the dialog metamodel and processes end-user actions – button clicks, values entered in text fields, etc. The tree engine constructs tree diagrams and processes end-user actions – clicking on tree elements, collapsing tree elements, etc.

Theoretically we can build DSML tools using only the presented engines and their metamodels, that is, for every single tool we have to create a list of model transformations that process user created events according to the tool specification. However, it is a laborious process and therefore the question, whether this process can be simplified, arises. The paper presents the solution to this problem and it proposes the following:

- to develop a formal DSML tool specification language;
- to develop the interpreter of this language as a list of universal transformations that create a working tool from the given specification;
- to develop the configurator to give the tool specification faster and easier.

2. Formal DSML tool specification

The purpose of this chapter is to find a formal specification language (or method), which can specify a wide range of DSML languages and their corresponding tools. If we managed to find such a formal language, then we can construct a universal interpreter (or compiler) that converts every specification (written in the language) into the working tool. Thus to get a working tool, it is sufficient to pass a particular DSML tool specification to the interpreter.

To better explain the complexity of the task, we will look at a simpler task, that is, we will seek a formal specification method, which can define a wide range of DSML languages (without their corresponding tools). The main problem is that there are well-known single-level metamodeling languages, that is, there are well-known techniques to define particular languages by metamodels, for example, UML activity diagrams, UML class diagrams, flowchart diagrams, etc., but there are no formal methods allowing rising one meta level higher, that is, to specify a family of languages instead of one particular language. In other words, a single-level metamodeling problem is well resolved, but the two-level metamodeling problem in the context of DSML languages is not.

The paper presents an alternative metamodeling approach, namely, the tool definition metamodel that is presented in Fig. 2. The tool definition metamodel describes two things – the tool language and its behavior. Tool languages are specified by the class *GraphDiagramType* and its attribute *caption* stores the language name. The elements of the language are specified by the class *ElemType* and its attribute *caption* stores the name of the element, but the composition *elemType-graphDiagramType* shows which language the element belongs. The class *ElemType* and its subclasses *NodeType*, *PortType* and *EdgeType* specify whether the element type is box, port or line.

Every element type has its constraints. The association *containerType-componentType* shows that one type of boxes may contain other type of boxes. The association *portType-nodeType* shows that ports always have to be attached to a certain type of box. But to define a line, we have to specify which element is the start element and which is the end element, and this information is given by two associations *start-eStart* and *end-eEnd*.

The association *subType-superType* shows that an element is a super type to another element, that is, the association allows making element type hierarchy where subtypes inherit its super type restrictions. In other words, this construction allows simulating the generalization relationship used in UML class diagrams.

In order to specify an element appearance, we have to add its style. The class *GraphDiagramStyle* specifies diagram's style, the class *ElemStyle* and its subclasses *NodeStyle*, *PortStyle* and *EdgeStyle* specify the style for boxes, ports and lines, but *CompartStyle* specifies attribute styles.

The other thing the tool definition metamodel describes is tool's behavior. To build new elements, the metamodel contains classes *PaletteType* and *PaletteElementType*. The association *graphDiagramType-paletteType* shows the language the palette belongs, but the association *elemType-paletteElementType* shows the corresponding element to the

palette button. The class *PaletteElementType* has attributes - *caption*, *nr* and *picture*. The attribute *caption* stores the button's name, *nr* stores the position of the button on the palette, but *picture* stores the address of the button's picture.

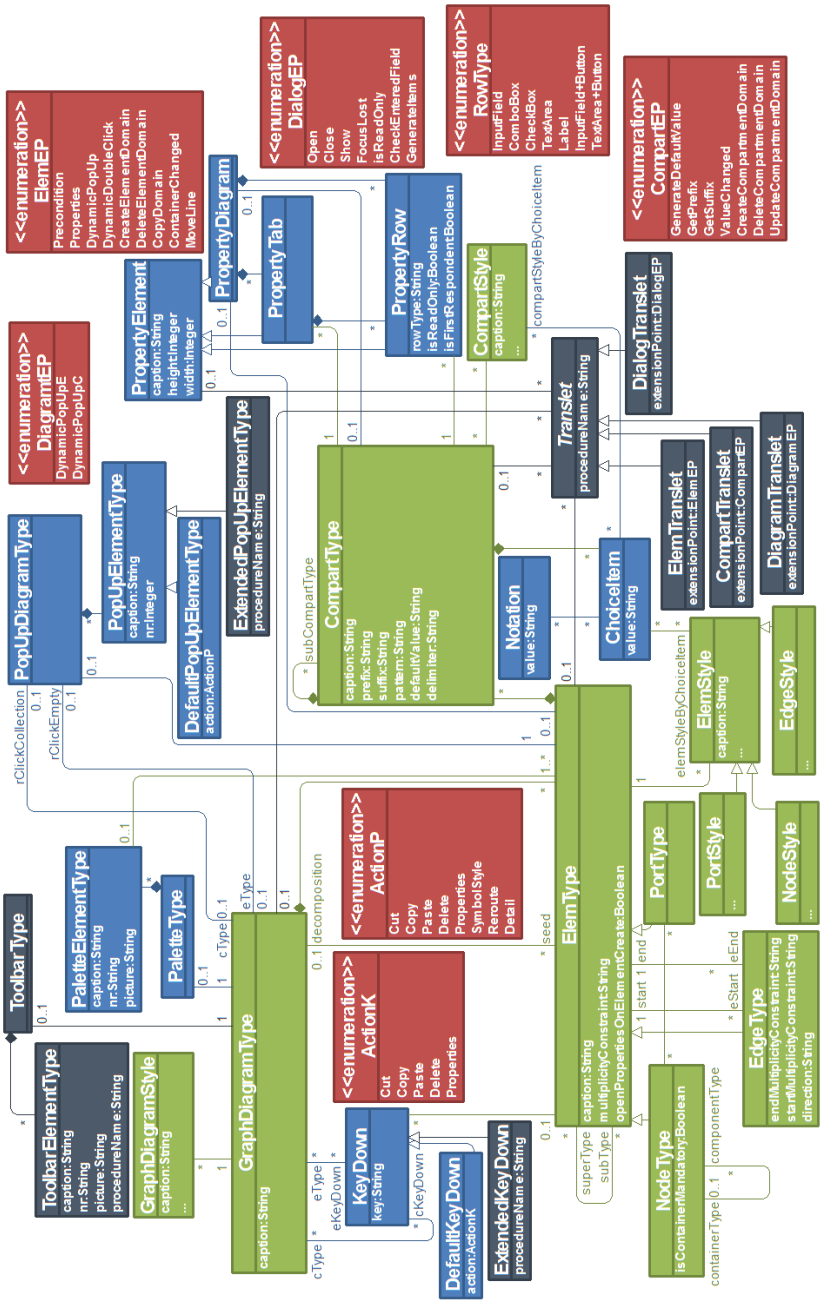

Fig. 2. Tool definition metamodel

To implement the context menu (appear when users perform mouse right-clicks), new classes and associations are introduced in the metamodel. The class *PopUpDiagramType* corresponds to the context menu, but its items are specified by *PopUpElementType* but its attributes *caption* and *nr* stores item's name and item's position on the menu. To provide items with predefined actions, the metamodel contains the class *DefaultPopUpElementType* and its attribute *action* respectively stores one of the predefined actions.

To display the context menu, three cases are distinguished depending on the context. In case the user has clicked on a single element, the context menu is constructed using the association *elemType-popUpDiagramType*. In case the user has clicked on an empty place in the diagram, the context menu is constructed using the association *rClickCollection-cType*. And finally, in case the user has clicked on an element collection, the context menu is constructed using the association *rClickCollection-cType*.

The class *KeyDown* and its attribute *key* specifies the pressed key combination. By analogy with context menus, to provide predefined actions, the metamodel contains the class *DefaultKeyDown* and its attribute *action* stores one of the predefined actions. But the context of the key combination is distinguished by the introduced three cases, namely, if a key combination is pressed when there is one active element (association *elemType-keyDown*), no active elements (association *eType-eKeyDown*) and several active elements (association *cType-cKeyDown*).

To enter attribute values, generic dialog windows are constructed. The class *PropertyElement* is an abstract class, which indicates that all dialog window components have an attribute *caption* to store its name. The class *PropertyDiagram* corresponds to the dialog form and it consists of rows or tabs (corresponding classes are *PropertyRow* and *PropertyTab*). Every dialogue window's row is displayed as a row having a name and an input field. The attribute *caption* (the attribute is inherited from the class *PropertyElement*) corresponds to the row name, but the input field type is specified by the attribute's *rowType* value. The class *PropertyTab* allows building tabs on dialogue windows, thus the rows can be located directly on the dialog window (specified by the association *propertyRow-propertyDiagram*) or on the tab (specified by the association *propertyRow-propertyTab*).

The purpose of classes *ToolbarType*, *ToolbarElementType*, *ExtendedPopUpElementType*, *ExtendedKeyDown* and *Translet* is to allow adding transformations that are specifically designed for the specific tool, thus supplementing the interpreter's behavior (see Chapter 3.2.).

3. Implementation

The previously described tool building metamodel is constructed in a way that all the necessary information that gives a DSML tool specification, is stored as the instance of the tool definition metamodel. This means that we can develop an interpreter (or a list of universal transformations) that creates a working tool from the given specification. Thus it reduces the tool development process to creation of instances of the tool definition metamodel.

3.1. Implementation metamodel

To apply the tool definition metamodel in practice, it has to be integrated with the existing platform metamodels, namely, presentation metamodel, dialog window metamodel and tree diagram metamodel. The combination of these four metamodels accompanied by scaffolding associations gives us the implementation metamodel as it is schematically represented in Fig. 3.

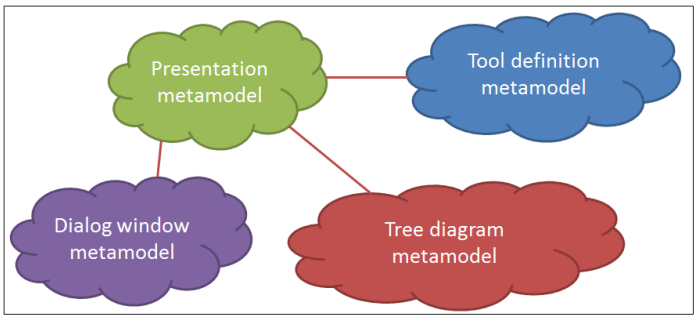

Fig. 3. Schematic representation of implementation metamodel

A simplified representation of the metamodel (scaffolding associations are colored thicker and they are between classes GraphDiagram and GraphDiagramType, Element and ElemType, Compartment and CompartType, Compartment and Component, GraphDiagram and Node) is shown in Fig. 4 (a simplified metamodel).

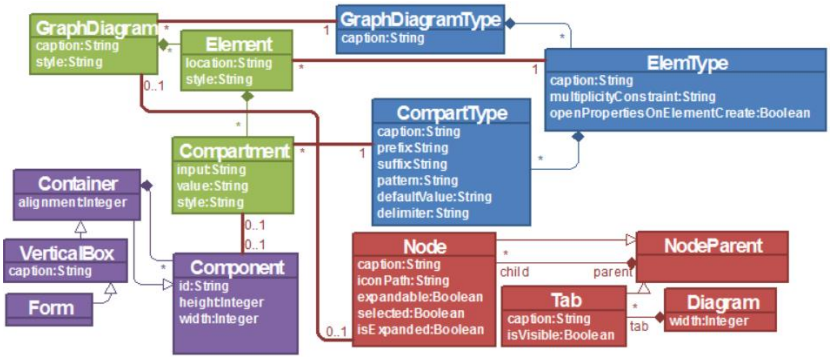

Fig. 4. A simplified implementation metamodel

3.2. Interpreter and extension points

As already mentioned before, to build a DSML tool means to implement a list of tool specific transformations. Since we have a universal tool definition metamodel, it allows us to process each platform's event by a universal transformation and thus every tool specific transformation can be replaced by the universal transformation.

To perform this task, we have to take a list of events of every platform's engine and then we have to implement universal transformations that process them. Since the platform has three engines, we have to take only the list of their events.

However, it is not possible to cover every single feature by a single metamodel and a fixed interpreter. Therefore an extension point mechanism is introduced allowing executing tool specific lQuery transformations during the run-time of the universal transformation. Since the platform's internal data structure is open, the scope of tool specific transformations is the whole repository (i.e., implementation metamodel) and they can arbitrarily change its content.

Extension points are divided into two categories - weak and hard extension points. Weak extension points occur when the user performs one of the following actions – presses a toolbar button, selects an operation from the context menu or enters a key combination. To execute these actions the metamodel contains classes *ToolbarElementType*, *ExtendedPopUpElementType* and *ExtendedKeyDown*, and its attribute *procedureName* stores the name of the transformation that will be executed when the user performs one of the mentioned actions.

The hard extension points allow supplementing universal transformations during the run-time and in the metamodel they are implemented through the class *Translet* and its subclasses *DiagramTranslet*, *ElemTranslet*, *CompartTranslet* and *DialogTranslet*. The attribute *extensionPoint* stores the extension point's name, but the attribute *procedureName* stores the transformation name.

Technically speaking, extension points are implemented as procedure (transformation) calls in certain places in universal transformations, and to precisely explain them, every universal transformation containing an extension point is represented by a flowchart which precisely indicates what steps have been made before the execution of the extension point transformation and what steps will be taken after.

To demonstrate the idea of hard extension points, the Fig. 5 shows a flowchart describing the universal transformation processing *RClick* event. Thus, when the user performs a right-click, *RClick* event is created and the universal transformation distinguishes three cases.

In case the user has clicked on a single element, the transformation searches the transformation corresponding to the extension point "DynamicPopUp", namely, the transformation looks for element's corresponding *ElemType* instance and then for *Translet* instance (link *elemType-translet*) that has an attribute *extensionPoint* containing „DynamicPopUp" value. If such an instance of *Translet* exists, then the universal transformation executes the transformation that is stored in the *Translet* instance's attribute *procedureName*. If not, the universal transformation constructs a static context menu from the instances of the tool definition metamodel.

In case the user has clicked on an element collection, the universal transformation searches the transformation corresponding to the extension point "DynamicPopUpC". The following steps are the same as in the first case – the context menu is dynamically built by the extension point transformation (if there is a corresponding *Translet* instance), or it is statically built from the instances of the tool definition metamodel (if there is no corresponding *Translet* instance).

In case the user has clicked on an empty place in the diagram, the transformation searches the transformation corresponding to the extension point "DynamicPopUpE" and then the context menu is built either dynamically or statically.

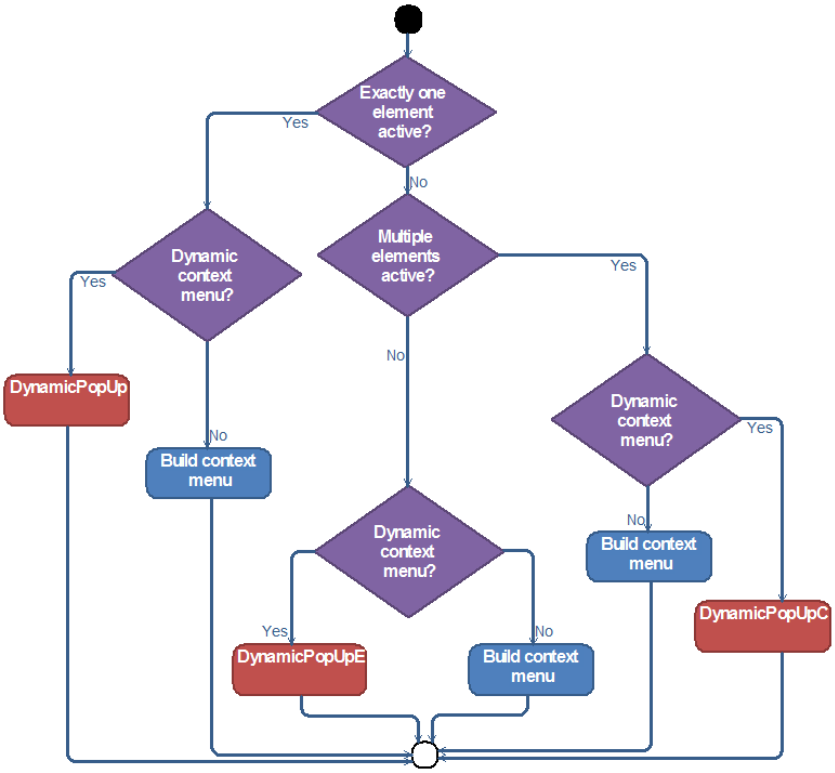

Fig. 5. *RClick* flowchart

4. Configurator

In the previous chapters we saw that DSML tools can be specified by the instances of the tool definition metamodel, and then processed by the interpreter to create a working tool from the given specification. Thus, the effectiveness of the DSML tool development process depends on how quickly and easily the instances of the type metamodel can be created.

To simplify and accelerate the creation of the tool definition metamodel instances, the configurator (Sprogis and Barzdins, 2013), (Sprogis et.al., 2010), (Sprogis, 2010), (WEB, d) was developed. The configurator is a DSML tool allowing specifying DSML tools graphically. The language used by the configurator is in a higher level of abstraction compared to UML metamodels, namely, they are more compact then UML class diagrams because element styles are defined by the look and feel and the relevant information is entered using dialogue forms.

The benefit of using the configurator is that it automatically transforms its models to instances of the tool definition metamodel, and the tool developer does not have to know the details of the tool definition metamodel. Thus, the application of configurator significantly reduces the time spent to create instances of the type metamodel and excludes the possibility to create instances of the type metamodel that is not supported by the interpreter.

4.1. Configurator language

The configurator's graphical language consists of four elements - *Box*, *Line*, *Port*, and *Specialization* (see Fig. 6). The element *Box* specifies the elements of box type, *Line* specifies the elements of line type, *Port* specifies the elements of port type, but *Specialization* shows that one element is a subtype to another (*Box1* is a subtype to *Box*).

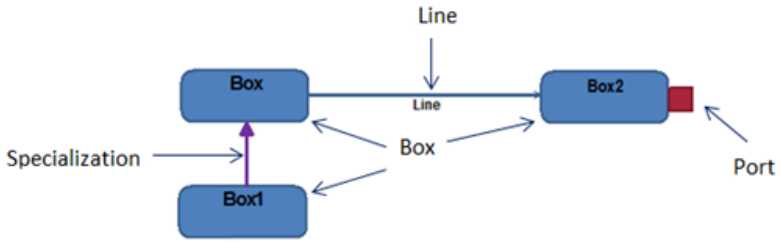

Fig. 6. Elements of the configurator language

These four elements allow defining DSML tools in the configurator. Fig. 7 shows the definition of the flowchart editor that is used to draw the flowchart diagram in Fig. 5. The definition indicates that elements - "Start", "Activity", "Branching" and "End" are of box type but elements "Out" and "In" are abstract super types showing the start and the end elements of "Flow". Since subtypes inherit incoming and outgoing lines of their super types, the definition specifies that the following pairs can be connected: "Start" and "Activity", "Start" and "Branching", "Start" and "End", "Activity" and "Activity", "Activity" and "Branching", "Activity" and "End", "Branching" and "Activity", "Branching" and "Branching", "Branching" and "End" (but the following pairs cannot be connected - "End" and "Start", "End" and "Activity", "End" and "Branching", "End" and "End", "Start" and "Start", "Activity" and "Start" and "Branching" and "Start").

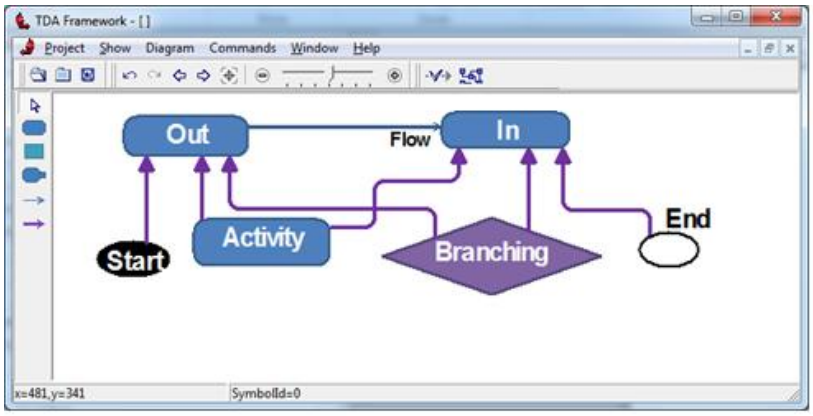

Fig. 7. The configuration of the flowchart editor

4.2. The implementation of the configurator

The configurator is implemented as a DSML tool using the components of TDA and exploiting the same mechanism as for the other DSML tools, namely, the specification

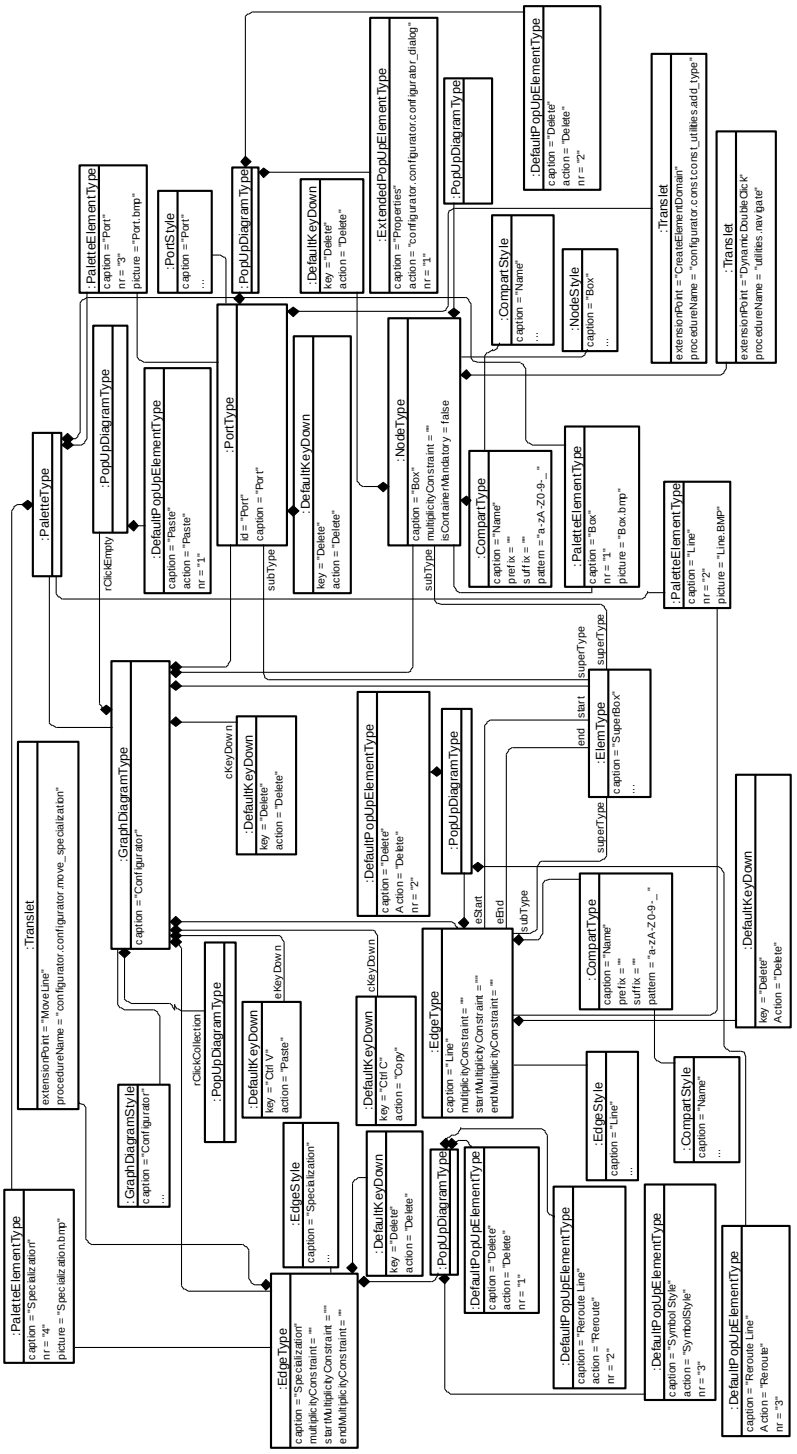

Fig. 8. A simplified configurator's specification

of the configurator is given by the instance of the tool definition metamodel, and its events are processed by the interpreter's transformations (although the transformation functionality is extended through the extension point mechanism).

As mentioned in chapter 4.1, then configurator's language consists of four graphical elements – *Box*, *Line*, *Port* and *Specialization* and thus to specify these elements, we have to create the specification with the tool definition metamodel instances, as shown in Fig. 8 (a simplified specification).

We should note, since the configurator is specified by the instances of the tool definition metamodel, it is possible to give the same specification by the configurator itself (bootstrapping method), and its corresponding specification (graphical model) by the configurator is shown in Fig. 9.

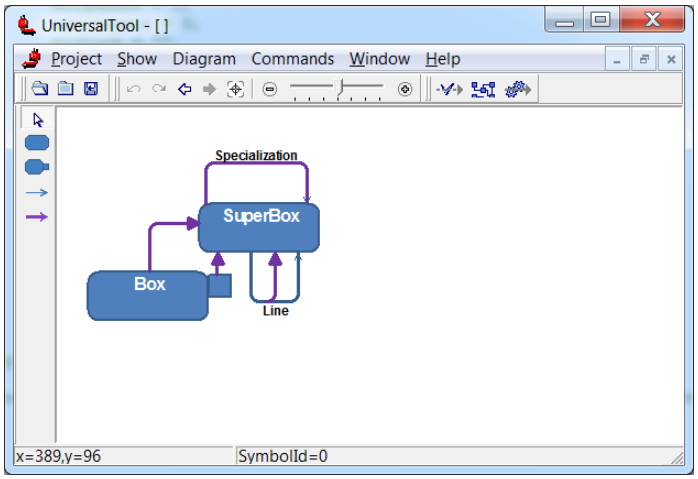

Fig. 9. The graphical model of configurator's specification

The next step is to interpret the given specification. In case of the configurator, the interpreter's functionality is not sufficient and therefore it is supplemented with six transformations in the following extension points - *Properties*, *DynamicDoubleClick*, *CreateElementDomain*, *DeleteElementDomain*, *CopyDomain* and *MoveLine*.

4.3. Tool versioning and model data migration

Typically the development of DSML tool is an iterative process falling under the following scenario. At the first step we develop the first version of the tool, and then we update it and produce the second version, the third, fourth, and so on until we achieve the desired result. The problem of such a development process is that there are models created by older tool versions and it is necessary to use them by the latest tool version. Thus, the platform needs a mechanism that allows migrating models created by the older tool version to the latest tool version.

At first, we should specify that by the "tool" and "version of the tool" we mean its specification by the tool definition metamodel instance, or, in other words, a particular tool version is an instance of the tool definition metamodel, but by "model data" we mean an instance of presentation metamodel. Thus, "to develop a new version of the tool" means to create an instance of the tool definition metamodel, but "to migrate model

data from the older version to the latest" means to achieve the situation that model data are linked to the new tool's specification, meaning, the presentation metamodel instance is linked to the latest tool definition metamodel instance. To achieve the desired situation, there are two possible scenarios. One possible scenario is to evaluate the old specification to the latest, the second scenario is to replace the old specification by the latest.

We will look at each scenario in more details. In order to execute evaluation scenario, we have to use two things. Firstly, the latest version is created by executing a sequence of actions in the older tool version. Secondly, tools are developed by the configurator, and it allows us to record user actions performed in the configurator. Thus, we can record all the user actions that were executed to create the latest tool version and use them to develop a transformation that migrates model data between two different tool versions.

In contrary, the second scenario replaces the old tool specification by the latest instead of evaluating it. The migration process according to the second scenario consists of two steps and the initial state of the repository before the migration has started is shown in Fig. 10 representing the instances of the tool definition and the presentation metamodels.

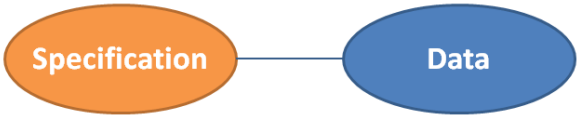

Fig. 10. The initial state of the repository

The first step creates new instance of the tool definition metamodel, thus there are old and new instances of the tool definition metamodel in the repository, but the presentation metamodel instance is still linked to the old instance of the tool definition metamodel. The result of the first migration step is shown in Fig. 11.

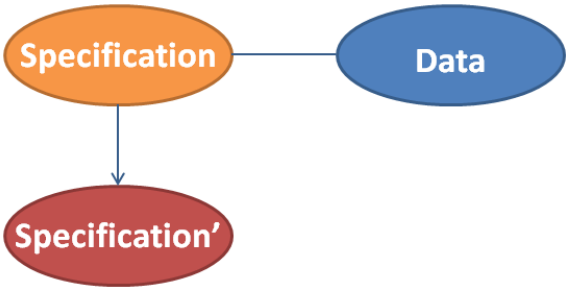

Fig. 11. The result of the first migration

The second step disconnects the instance of the presentation metamodel from the old instance of the tool definition metamodel, then links them to the new instance of the tool definition metamodel and finally deletes the old instance of the tool definition metamodel from the repository. The result of the second migration step is shown in Fig. 12.

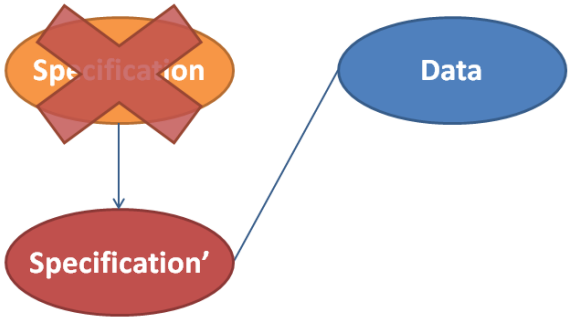

Fig. 12. The result of the migration process

By comparing the two scenarios, we can see that evaluation scenario requires recording all the actions that were performed to create the next tool version. In other words, we have to store every single change in the tool definition metamodel instances, and taking into account the size of the tool definition metamodel, the implementation of the evaluation scenario would be complex. Whereas executing the replacement scenario we have to migrate all the links so that they connect model data and the latest tool specification. Since the tool definition metamodel and the presentation metamodel are connected by only three associations, then it is easier to track the changes in the tool definition metamodel instances to reconstruct links of the mentioned three associations then to track all the changes in the tool definition metamodel, and, therefore, the replacement scenario is used for implementation of the tool versioning and model data migration.

5. Practical applications

The proposed method of defining DSML tool has been validated in practice, and it has developed a number of tools.

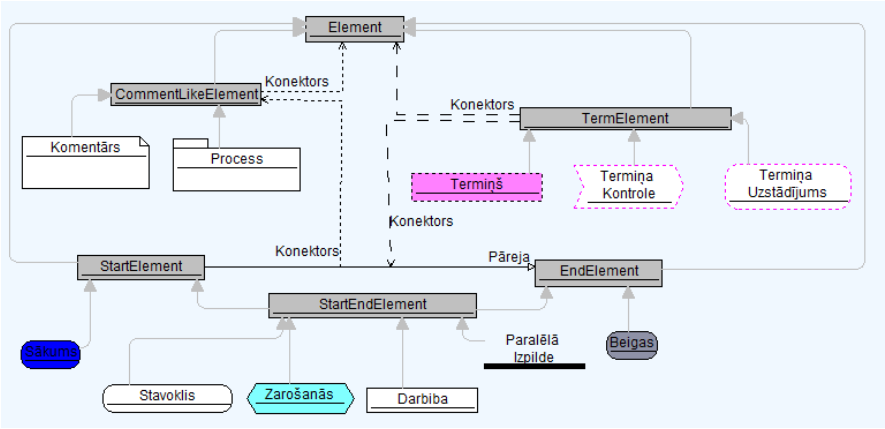

Fig. 13. The graphical model of BiLingva configuration

5.1. BiLingva

BiLingva (Karnitis et.al., 2009) is a business process modeling tool developed to model processes for Latvian Investment and Development Agency. The graphical model of tool configuration is shown in Fig. 13.

5.2. ProMod

ProMod (Barzdins et.al., 2009a) is a business process modeling tool developed to model processes for the State Social Insurance Agency. The graphical model of configuration for one diagram type that is implemented in the tool is shown in Fig. 14.

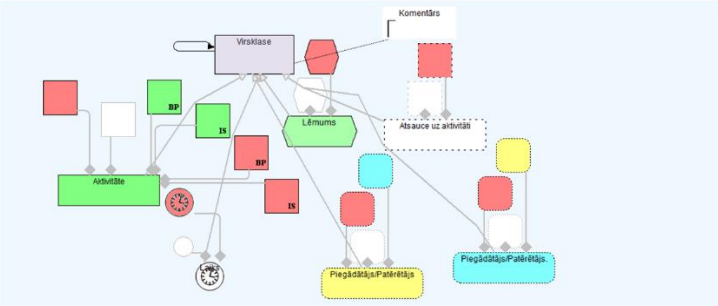

Fig. 14. The graphical model of ProMod configuration

5.3. OWLGrEd

OWGrEd (WEB, j), (Liepins et.al, 2012), (Cerans et.al., 2012), (Cerans et.al., 2013), (Barzdins et.al., 2010), (Barzdins et.al., 2011) is a UML class diagram syntax-based graphical OWL ontology (WEB, i) editor. The graphical model of tool configuration is shown in Fig. 15.

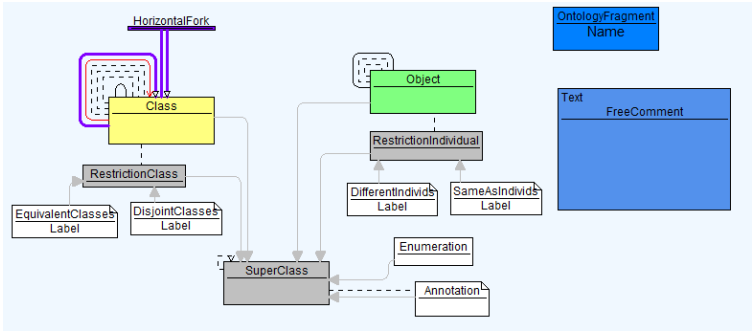

Fig. 15. The graphical model of OWLGrEd configuration

5.4. LUMod

LUMod (Barzdins et.al., 2013) is a business process modeling tool developed to model processes for University of Latvia. The graphical model of tool configuration is shown in Fig. 16.

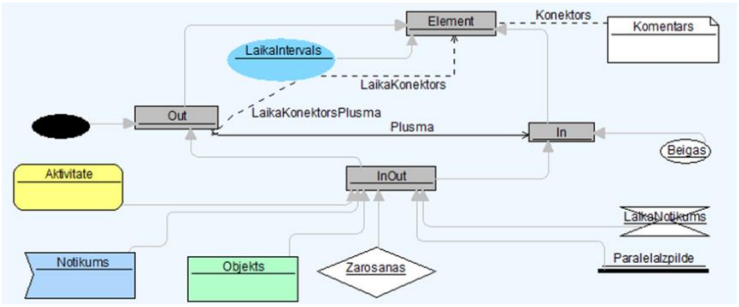

Fig. 16. The graphical model of LUMod configuration

6. Related work

Graphical tool building is not a new topic, and therefore there are a number of existing tool building platforms. Typically these platforms require, at first, to specify an abstract syntax model of the language using UML class diagrams (Eclipse GMF, Microsoft DSL, ObeoDesigner (WEB, h), Graphiti (WEB, e)), UML profile (RSA (WEB, f)) or the language that is similar to UML class diagrams (GME (Davis, 2003), (WEB, c)). When abstract syntax model is specified, we have to specify the concrete syntax of the language, and usually it means to create the concrete syntax model and then map it to the abstract syntax model. If the tool we are developing requires an additional functionality, it is possible to program it, but mostly it is not a simple task requiring to intervene in the platform's internal structure or in the generated code. The conclusion is that such a tool specification process is long, resulting in complicated and difficult to understand tool development process.

A different tool building approach is used by MetaEdit allowing specifying the tool's language directly in terms of concrete syntax, thus significantly reducing the complexity of the tool development process. The conclusion is that this is a convenient approach to develop tools but MetaEdit solution has a very limited ability to complement the tool's functionality.

The main difference between the proposed approach and the existing platforms is that the proposed approach allows a specification of the tool language in terms of concrete syntax (that is, configurator's graphical model) and, if necessary, an addition of the tool specific functionality using the extension point mechanism. Besides, the extension point mechanism allows not only an addition of toolbar buttons, context menus, etc. (through weak extensions) but it also allows intervening in the interpreter's run-time with tool specific transformations (through hard extensions). Secondly, the proposed approach allows migrating tool models between different tool versions.

We have to admit that the proposed approach has its limits. Firstly, the proposed approach can define such languages that are composed of three types of elements –

boxes, lines and ports. Secondly, the tools build using the proposed approach do not offer such a wide range of graphical elements and a rich interface as, for instance, Microsoft Visio (WEB, o) does.

7. Conclusions

The purpose of the paper is to present a DSML tool building approach that is relatively easy, allows extending the platform's functionality and is open for external applications. To accomplish the task, the proposed approach uses the TDA platform components. Generally speaking, it would be enough to build a DSML tools using these components, namely, for every single tool we would have to create a list of model transformations that process user created events. However, it is a laborious and complex process, and therefore the paper presents a solution that proposes the following three issues:

- development of a formal DSML tool specification language;
- development of the interpreter of this language that transforms the given specification in a working tool;
- development of the configurator to make the tool specification process faster and easier.

To accomplish the first task, that is, to develop a formal DSML tool specification language the tool definition metamodel was created.

To transform a specification in a working tool, a universal interpreter was developed as a list of universal transformations that universally process user created events. In addition, to be able to go beyond the fixed functionality of the interpreter, an extension point mechanism was introduced allowing executing tool specific transformations during the run-time of the universal transformation. It should be noted that the platform's internal data structure is open, thus the transformations can easily access model data and modify them. As a result, we can use the interpreter to develop tools of different complexity, ranging from as simple as a flowchart editor, that can be implemented without programming, and as complex as the configurator that is powerful enough to develop other DSML tools.

To simplify the creation of instances of the tool definition metamodel, the configurator tool was developed allowing specifying DSML tools graphically, not in the level of the tool definition metamodel. The purpose of the configurator is to facilitate the DSML tool specification process, and the main idea is to specify DSMLs "directly" by the concrete syntax model instead of abstract syntax model. Thus, unlike most other platforms, which use both models to specify the language and then establish mappings between them, the configurator specifies only one model and no mappings. Since there is a broad class of DSML tools that do not require performing complex calculations and model processions, the proposed method is suitable as it reduces the development time and simplifies the development process.

Another advantage of the configurator is that, when it is combined with the interpreter, they form a technology that significantly increases tool developer's productivity by making the tool development process iterative with the following advantages. Firstly, the language concepts are defined one-by-one, thus the obtained result can be checked after each step. Secondly, we can modify and run Lua (interpreted language) programs (transformation are implemented in Lua (Ierusalimschy, 2006)) without restarting the tool and no compilation. Thirdly, perform model migrations between different tool versions.

The described technology has been practically verified and successfully applied to develop such tools - ProMod, BiLingva, LuMod and OWLGrEd.

Acknowledgement

This work has been supported by the European Social Fund within the project "Support for Doctoral Studies at University of Latvia" and ESF project 2013/0005/1DP/1.1.1.2.0/13/APIA/VIAA/049.

References

Barzdins, J., Barzdins, G., Cerans, K., Liepins, R., Sprogis, A. (2010) UML Style Graphical Notation and Editor for OWL 2, Lecture Notes in Business Information Processing, Volume 64, Springer, 102-114, 2010 (SCOPUS)

Barzdins, J., Cerans, K., Kalnins, A., Grasmanis, M., Kozlovics, S., Lace, L., Liepins, R., Rencis, E., Sprogis, A., Zarins, A (2009a). Domain Specific Languages for Business Process Management: a Case Study, Proc. of Workshop on Domain-Specific Modeling, OOPSLA 2009, Nashville, USA

Barzdins, J., Cerans, K., Kozlovics, S., Rencis, E., Zarins, A (2009b). A Graph Diagram Engine for the Transformation-Driven Architecture. Proceedings of MDDAUI'09 Workshop of International Conference on Intelligent User Interfaces 2009, Sanibel Island, Florida, USA, pp. 29–32, 2009.

Barzdins, J., Cerans, K., Liepins, R., Sprogis, A. (2011) Advanced ontology visualization with OWLGrEd, publicēta OWLED 2011, OWL: Experiences and Directions, 8th International Workshop, San Francisco, California, USA

Barzdins, J., Rencis, E., Kozlovics, S. (2008) The Transformation-Driven Architecture, The 8th OOPSLA Workshop on Domain-Specific Modeling, October 19 - October 20, 2008, Nashville, TN.

Barzdins, J., Rencis, E., Sostaks, A. (2013) Towards Human-Executable Business Process Modeling. Frontiers in Artificial Intelligence and Applications, Volume 249: Databases and Information Systems VII, pp. 149-163, IOS Press, 2013 (SCOPUS)

Barzdins, J., Zarins, A., Cerans, K., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A. (2007) GrTP: Transformation Based Graphical Tool Building Platform, Proc. of Workshop on Model Driven Development of Advanced User Interfaces, MODELS 2007, Nashville, USA (SCOPUS)

Cerans, K., Barzdins, J., Barzdins, G., Liepins, R., Ovcinnikova, J., Rikacovs, S., Sprogis, A. (2012) Graphical Schema Editing for Stardog OWL/RDF Databases using OWLGrEd/S, CEUR-WS.org, 2012

Cerans, K., Liepins, R., Ovcinnikova, J., Sprogis, A. (2013) Advanced OWL 2.0 Ontology Visualization in OWLGrEd, Frontiers in Artificial Intelligence and Applications, vol. 249, IOS Press, pp. 41-54, 2013 (SCOPUS)

Cook, S., Jones, G., Kent, S., and Wills, A. C. (2007) Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007

Davis, J. (2003) GME: the generic modeling environment, OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2003

Ierusalimschy, R. (2006) Programming in Lua. Lightning Source UK Ltd., Milton Keynes, UK, 2006

Karnitis, G., Bicevskis, J., Cerina-Berzina, J. (2009) Information systems development based on visual Domain Specific Language BiLingva, Proceedings of 4th IFIP TC2 Central and

Eastern European conference on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland

Kelly, S., Tolvanen, J.-P. (2008) Domain-Specific Modeling: Enabling Full Code Generation, Wiley, 2008, p. 448.

Kozlovics, S. (2012). The Transformation-Driven Architecture and its Graphical Presentation Engines, PhD thesis, University of Latvia, Latvia.

Kozlovics, S. (2010) A Dialog Engine Metamodel for the Transformation-Driven Architecture. Computer Science and Information Technologies, Scientific Papers University of Latvia, Vol. 756, pp. 151-170., 2010

Liepins, R. (2012) Library for model querying: IQuery. Proceedings of the 12th Workshop on OCL and Textual Modelling, pp. 31-36., ACM, 2012.

Liepins, R. (2011) lQuery: A Model Query and Transformation Library. Computer Science and Information Technologies, Scientific Papers University of Latvia, 2011, Vol. 770, pp. 27-46.

Liepins, R., Cerans, K., Sprogis, A. Visualizing and Editing Ontology Fragments with OWLGrEd, I-SEMANTICS Poster and Demo Track, 2012

Robert, S., Gerard, S., Terrier, F., Lagarde, F. (2009) A Lightweight Approach for DomainSpecific Modeling Languages Design, Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on, pp.155-161, 27-29 Aug. 2009.

Sprinkle, J., Rumpe, B, Vangheluwe, H., Karsai, G. (2007) Metamodelling - State of the Art and Research Challenges. Model-Based Engineering of Embedded Real-Time Systems 2007: 57-76

Sprogis, A. (2010) The Configurator in DSL Tool Building, Scientific Papers, University of Latvia, 2010. Vol. 756 Computer Science and Information Technologies, 173–192 p

Sprogis, A., Barzdins, J. (2013) Specification, Configuration and Implementation of DSL Tool, Frontiers in Artificial Intelligence and Applications, vol. 249, IOS Press, p. 330-343, 2013 (SCOPUS)

Sprogis, A., Liepins, R., Barzdins, J., Cerans, K., Kozlovics, S., Lace, L., Rencis, E., Zarins, A. (2010) GRAF: a Graphical Tool Building Framework, ECMFA 2010 Tools and Consultancy track, 2010, Paris, France

WEB (a) BPMN, http://www.bpmn.org

WEB (b) Eclipse Modeling, http://www.eclipse.org/modeling/emf/

WEB (c) GME: Generic Modeling Environment | Institute for Software Integrated Systems, http://www.isis.vanderbilt.edu/Projects/gme

WEB (d) Grade2 – Graphical Tool Building Framework, grade2.lumii.lv

WEB (e) Graphiti Home, http://www.eclipse.org/graphiti/

WEB (f) IBM Rational Software Architect, http://www.ibm.com/developerworks/rational/products/rsa/

WEB (g) MetaCase - MetaEdit+ Modeler DSM Tool, http://www.metacase.com/mep/

WEB (h) Obeo Designer: Domain Specific Modeling for Software Architects, http://www.obeodesigner.com/

WEB (i) OWL 2 Web Ontology Language Document Overview, http://www.w3.org/TR/owl2-overview/

WEB (j) OWLGrEd – Editor for Compact UML-style OWL Graphic Notation, owlgred.lumii.lv

WEB (k) SysML, http://www.omgsysml.org/

WEB (l) UML, http://www.uml.org

WEB (o) Visio Professional 2013, visio.microsoft.com

WEB (p) Visualization and Modeling SDK (DSL Tools), http://code.msdn.microsoft.com/windowsdesktop/Visualization-and-Modeling-313535db

Received May 16, 2014, accepted June 2, 2014