Baltic J. Modern Computing, Vol. 4 (2016), No. 4, 910-933
http://dx.doi.org/10.22364/bjmc.2016.4.4.20

Metamodel Specialization for Graphical Language and Editor Definition

Janis BARZDINS, Audris KALNINS

Institute of Mathematics and Computer Science, University of Latvia,
Raiņa bulvaris.29, Riga, LV-1459

audris.kalnins@lumii.lv, janis.barzdins@lumii.lv

Abstract. This paper is a further development of ideas presented by the authors in the previous papers on this topic. More concretely, in this paper three ideas have got a further development. First, the usage of metamodel specialization for graphical language and editor definition will be explained in a more detailed way. Second, a more general universal metamodel (UMM) will be offered, which covers a richer class of graphical languages. Third, the offered universal metamodel will enable a fluent transition from graphical language definition to their graphical editor definition, UMM for graphical tool definition will be a relatively slight extension of UMM for language definition. Finally, the paper presents also basic ideas for implementation of a graphical tool building platform based on the proposed metamodel specialization approach.

Keywords. Metamodeling, metamodel specialization, graphical syntax definition, graphical DSL, graphical editors.

Introduction

The paper is mainly devoted to the research domain which is now named *language workbenches* – frameworks for complete support of various kinds of domain specific languages (DSLs). This domain has been relatively widely investigated in the last twenty years, with many such workbenches developed and widely used in practice. At the beginning of the 2000s such tools were typically named metaCASE tools, but due to the recent massive transition from traditional CASE tools for software development to the more general concept of software development using DSLs the term *language workbench* is much more appropriate.

Currently there are numerous workbenches for the complete support of graphical DSLs. They support both the development of such languages by DSL designers and creation of a complete support for a given language including graphical editor and the language execution via compilers or interpreters. Therefore most of them are based on the abstract syntax (domain metamodel) definition of the language via the classic MOF approach as the first step and then adding a mapping from this syntax to graphical

notation elements for defining the diagram editors. First and foremost, these workbenches are based on Eclipse GMF (WEB, a), where the abstract syntax is defined via an EMF (WEB, b) metamodel, elements of concrete graphical syntax via GEF metamodel and mapping between these models via mapping metamodel. While the approach is quite universal, the practical usage is not so easy (Juliot, 2009). Therefore several improvements of the approach, such as Obeo Designer (Juliot, 2009, WEB, c), Sirius (WEB, d) or Eugenia (WEB, e) are offered. Another very popular workbench is MetaEdit (Kelly and Tolvanen, 2008) which is based on a domain specific metamodeling language GOPPRR (Graph-Object-Property-Port-Role-Relationship) supporting a mix of domain concepts, graphical notations and tool-related elements. A very pragmatic solution is the Microsoft DSL (Cook et al., 2007) workbench which also uses domain and graphics metamodels and a mapping between them. Certainly, there are much more similar workbenches, both commercial and open source. See a more detailed analysis of the related research in our paper (Kalnins and Barzdins, 2016 b). All these workbenches have as a primary goal the implementation of the given graphical language either by code generation or interpretation. Therefore for simpler use cases, when the main goal is just to create syntactically correct diagrams in the given graphical language, simpler and more direct solutions are possible. The platform devoted most directly to graphical DSL editor definition on the basis of the graphical syntax is the platform developed by IMCS UL – Transformation Driven Architecture (TDA) (Barzdins et al., 2007, 2008, Sprogis, 2010, 2013).

A common feature of all these workbenches is that they are based on a metamodel instantiation, typically in a MOF (OMG, 2015 c) style 4-layer architecture. The workbench itself has a fixed meta-metamodel, a tool for a given language (including both the language definition, and editor and other components) is based on a metamodel obtained by the instantiation. A model created by the tool user (in fact, a program in the given DSL) is obtained by instantiation of this metamodel, and at runtime the instantiated objects are present.

Now let us explain in more detail the instantiation approach. Let us take the TDA and consider the graphical language definition by instantiation. According to TDA principles this task is based on a Type metamodel for graphical languages. Fig. 1 shows such a simplified type metamodel, to be named further simply a Type metamodel in TDA. It is a fixed metamodel which contains type classes for all elements of a simple graphical diagram language – GraphDiagram, Node, Edge and Compartment (of a Node or Edge). In our approach the Compartment is any logical textual element in a diagram, a line or multiline (thus our terminology slightly differs from that used e. g. in Eclipse GMF (WEB, a) where only multiline texts are named compartments). The classes of

Type metamodel contain also the basic style attributes of diagram elements. A node compartment may be a structured text, e.g. a text line for a class attribute consists of attribute name, type, initial value etc., with relevant separator strings included. This structuring is supported by the relevant attributes and associations of the NodeCompartmentType class. The Type metamodel in TDA is, in fact, at MOF level M1

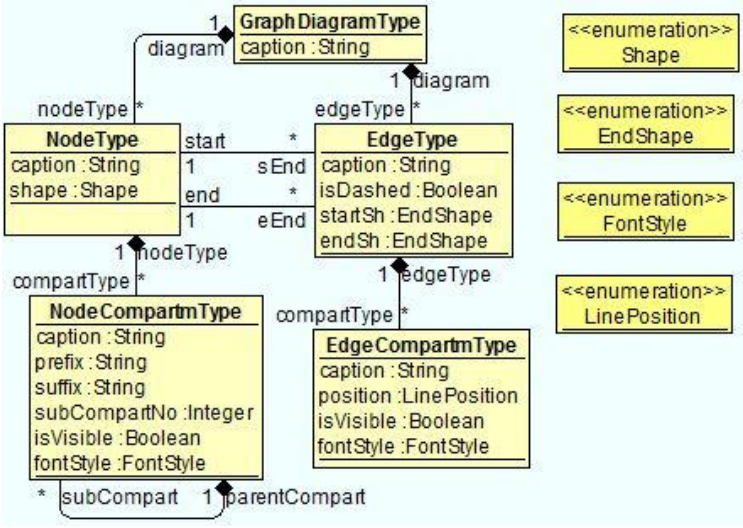

Fig. 1. Type metamodel in TDA

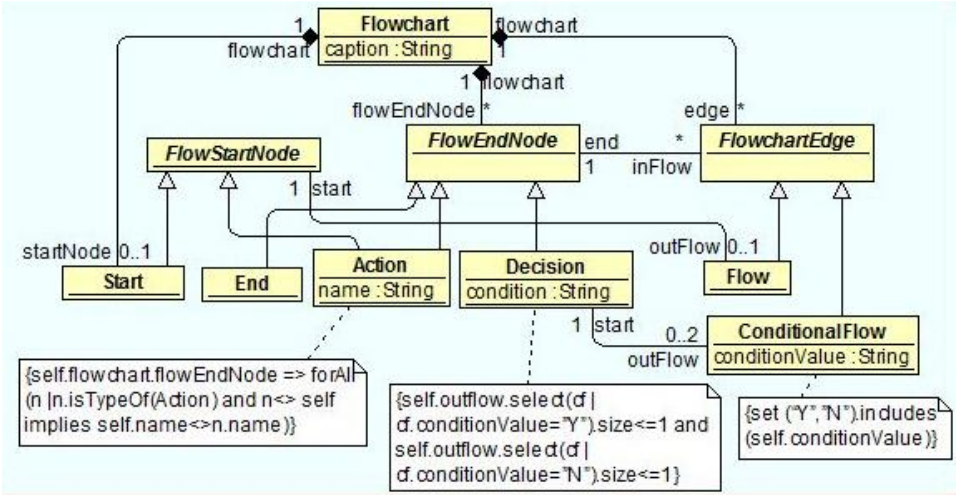

Fig. 2. Flowchart definition by a simple class diagram

– it is a fixed class diagram. The syntax for a concrete diagram notation is defined as an instance of this model – it is a UML object diagram. It is assumed that each node in a diagram has one of the defined node types, each edge has one of the edge types and so on.

The given Type metamodel supports the definition of a relatively wide class of graphical languages by means of instantiation. Let us illustrate this on the Flowchart diagram example. By Flowchart diagram we understand a graphical diagram which is defined by the EMOF-level model in Fig. 2.

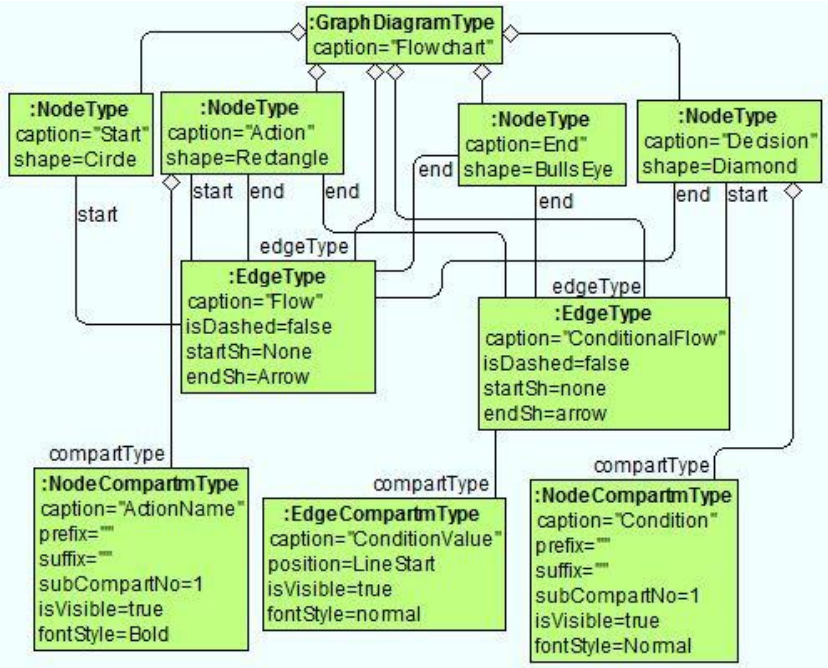

Fig. 3. Flowchart syntax definition by instantiation in TDA

In this diagram there are four node types – start node, action node, decision node and end node, and two edge types – flow and conditional flow. An action node contains a text – the action name, and decision node – the condition. A conditional flow also has a text attached – the condition value "Y" or "N" (for Yes or No). Other flowchart elements have no texts. There is a restriction that no more than one flow can start from a start or action node, but no more than two conditional flows can start from a decision node. Any number of flows or conditional flows can enter a node (except a start node). And there

may be only one start node per flowchart. These restrictions are shown in Fig. 2 by association multiplicities - appropriate abstract superclasses are introduced for this goal. Three simple OCL (OMG, 2012) constraints are added to express some semantic correctness rules for a diagram, e.g. that Action names must be unique per diagram.

The syntax of such Flowchart diagram can be specified by means of an instantiation of the Type metamodel in Fig. 1, see Fig. 3.

However, only the basic structure of the intended flowchart syntax can be defined in the object diagram in Fig. 3 – which edges can start from which nodes, what texts are associated to the diagram elements. But the permitted element multiplicities cannot be defined this way, and OCL constraints cannot be added. In the instantiation approach shown in Fig. 3 the only way for specifying constraints is to use a custom constraint language. This problem becomes even more serious, if we try to define tools for the language support.

In papers (Kalnins and Barzdins, 2016 a, b, c) the authors have offered a new approach based on metamodel specialization. The central element of this approach is a Universal Metamodel (UMM) which is being specialized for a concrete language, for instance flowcharts. This approach permits to use all UML class diagram features and OCL constraints for defining graphical languages and their support tools (graphical editors).

This paper is a further development of ideas presented by the authors in papers (Kalnins and Barzdins, 2016 a, b, c). First, the basis of the approach – the metamodel specialization is explained in a more detailed and understandable way. Second, the core of the approach – the universal metamodel to be specialized for the given graphical language is extended to cover a more general class of graphical languages. Third, the related nature of both tasks is more clearly emphasized. It is shown that the universal metamodel for the language editor definition is only a slight extension of the metamodel for syntax definition, since some attributes to be used for details of user interaction specification must be added.

The first section of the paper explains in detail the basic technology of the approach, namely, the metamodel specialization. The next section presents the core concept – the universal metamodel (UMM) for each of the tasks. Section 3 explains on examples how a graphical syntax of a language is defined by a UMM specialization. Section 4 extends the approach for an editor definition, and the concept of Universal Engine is explained in detail. Finally, Section 5 provides the basic principles how an editor definition workbench could be built using the proposed approach.

1. Metamodel specialization approach

Class specialization – creation of subclasses – is a well-known concept in UML. In a sense, it is a cornerstone in building understandable class diagrams. It is also a widely used approach in building metamodels in MOF (OMG, 2015 c). However, there is a

variation of specialization which can provide a new idea in building class models. It is the specialization of a whole metamodel.

The distinguishing feature in our approach is that only the created set of subclasses together with a set of related redefinitions of attributes and associations – the specialized metamodel is used for the given task.

A very important difference from the metamodel instantiation is that the specialized metamodel is at the same MOF meta-layer as the original one. This fact permits to use the same UML facilities as in the original metamodel, for example, OCL constraints.

To make the specialization process simpler and more readable, only a restricted set of UML class specialization facilities is used in our approach. The used metamodel specialization facilities include:

- Create subclasses of the source metamodel
- Redefine attributes – add new default values, but do not redefine attribute names, types and multiplicity (therefore no explicit *redefine*s modifier is needed here)
- Redefine association ends – names and multiplicity, explicit *redefine*s modifier is needed
- Add new OCL constraints to classes and attributes
- Do not add new (non-redefined) attributes and associations to subclasses

These restricted specialization facilities are sufficient for diagram syntax and editor definition and make the result compact and easy readable. For example, the addition of default values to attributes in the specialization is very natural for editor definition since these values are assigned at the class instance creation – a basic action in an editor. A specialization of a universal metamodel class may also be an abstract one (with the standard UML semantics), if it has a further specialization to non-abstract classes. This construct is typically used to simplify the association redefinition.

Since the concept of subclass is well-known from the very beginning of UML, certainly there are some known use cases of metamodel specialization. The most important such case is the OMG standard for Diagram Definition (DD) (OMG, 2015 b). There the Diagram Interchange (DI) metamodel is being specialized to its version for the given modeling notation, e.g. UMLDI. The full set of UML specialization facilities is used there because the original DI metamodel is at a very high abstraction level. Some other use cases are related to the support of DSL extension, and they are completely unrelated to the topic discussed in this paper.

2. Universal Metamodel for graphical syntax and editor Definition

In general, the starting point for the application of the specialization approach to a modeling task is the Universal Metamodel (UMM) for this task. However, as already mentioned, we try to make UMMs for both tasks discussed in this paper as similar as possible – the UMM for editor definition is only a slight extension of the UMM for

graphical syntax definition of the language. Therefore in this section we, in fact, provide one common metamodel by precisely denoting which elements of this metamodel are extensions for editor definition.

The intended application domain of our approach is graphical modeling languages with typical graph structure diagrams, consisting of nodes and edges with text elements added to both. However, some additional features are also supported, e.g. node nesting within another node.

Fig. 4 shows this metamodel. For both use cases of the metamodel the same set of classes is used. The attributes used only for editor definition are in bold font, for graphical syntax definition they should be ignored. Similarly, for editor definition two new enumerations are added – they have bold outlines in Fig. 4.

Now some comments on the classes of this metamodel are given. They reveal the general intention of the metamodel elements. Certainly, since according to our approach only specializations of this metamodel are used for graphical syntax (or editor) definition of a language, the precise semantics of metamodel elements will be explained on specialization examples in the following sections.

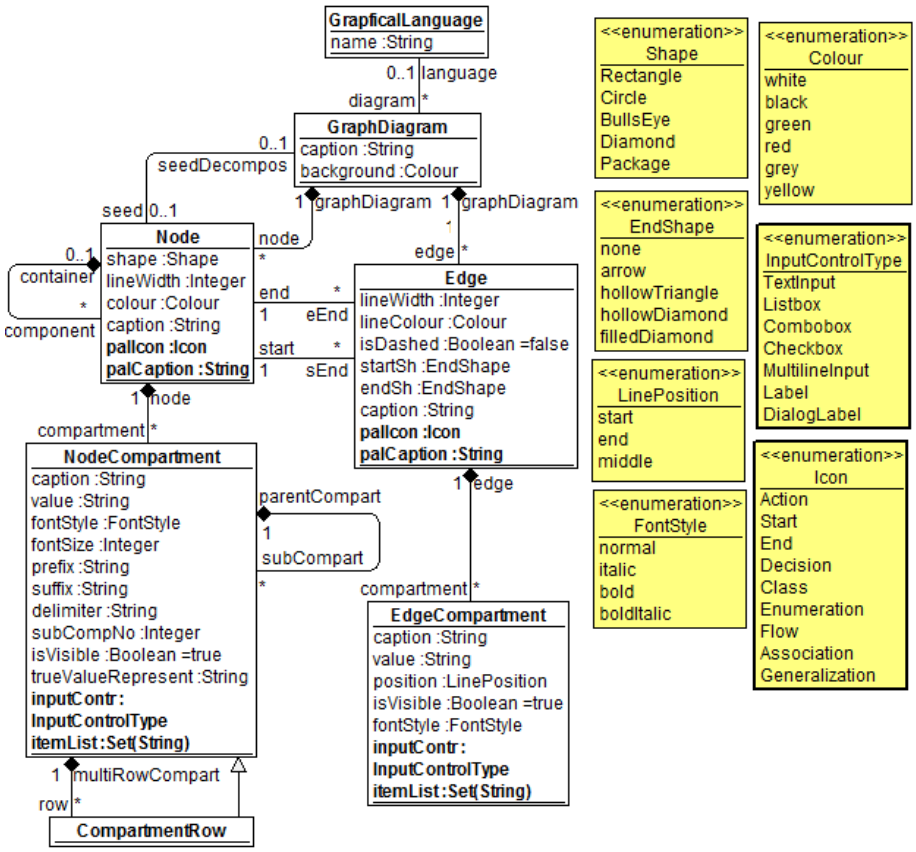

Fig. 4. UMM for graphical syntax and editor definition

The classes of the metamodel represent the concepts of a typical graph diagram – the graph diagram itself, nodes and edges in it and how they can be related. Both compartment classes define the structure of texts for nodes and edges respectively. The top class GraphicalLanguage represents a language which can contain several diagram kinds (as UML does). From the structure point of view this metamodel is similar to the type metamodel in Fig. 1, though their usage is completely different. A node compartment may be a structured text. Its parts (e.g. attribute name, type, default value etc.) are attached via the parentCompart – subCompart association, the order is defined via the subComprtNo attribute, a nested containment is supported as well. Prefixes and suffixes represent fixed text elements (separators, keywords etc.) to be inserted at appropriate places. Thus the complete logical structure of a compartment text can be defined – how the visible textual value of a compartment is assembled from its parts. In fact, this definition is semantically equivalent to the text structure definition by a context-free grammar such as EBNF (Grune et al., 2012) – as typically it is done for structured texts in graphical language specifications. The CompartmentRow class represents another kind of text structuring – a compartment consisting of similar rows – as it is for attributes in a class node. See a detailed text structuring example in Section 4 where a simple class diagram editor is defined.

The described facilities for diagram structure definition – both at diagram graphics level for possible node-edge relations and at text structuring level are used both for diagram syntax definition and editor definition. For diagram syntax definition they specify how a syntactically correct diagram should look like. But for editor definition they specify how a correct diagram should be created. Certainly, the detailed semantics of the approach will become clear only in the next sections where UMM specialization examples for both use cases will be explained in detail.

The additional attributes for editor definition (shown in bold in Fig. 4) define the way how the user interaction with the editor should occur. The palette-related attributes will be used to define a palette for a diagram kind – what elements it should contain and how they should look like. The inputContr attribute for compartments is used to specify, what kind of an input control should be used to enter this text (or part of a text). The complete structure of a dialog form for entering a text for a graphical diagram element in fact is determined by the text structuring facilities already explained. Certainly, the general behavior of such editor will become clear only when the second component of the editor – the Universal Engine will be explained. This will be done in Section 4 on the basis of UMM specialization examples for editors.

3. Graphical Language Definition by Metamodel specialization

In this section we show how the specialization of the Universal Metamodel introduced in the previous section is used for the precise graphical syntax definition of languages. We remind that only non-bold attributes in the UMM in Fig. 4 are used for this purpose. Fig. 5 presents the specialization of the UMM (in standard UML notation) for syntax

definition of the same flowchart diagram which was discussed in the introduction. Classes of UMM there have a white background, but specialized classes – a colored one.

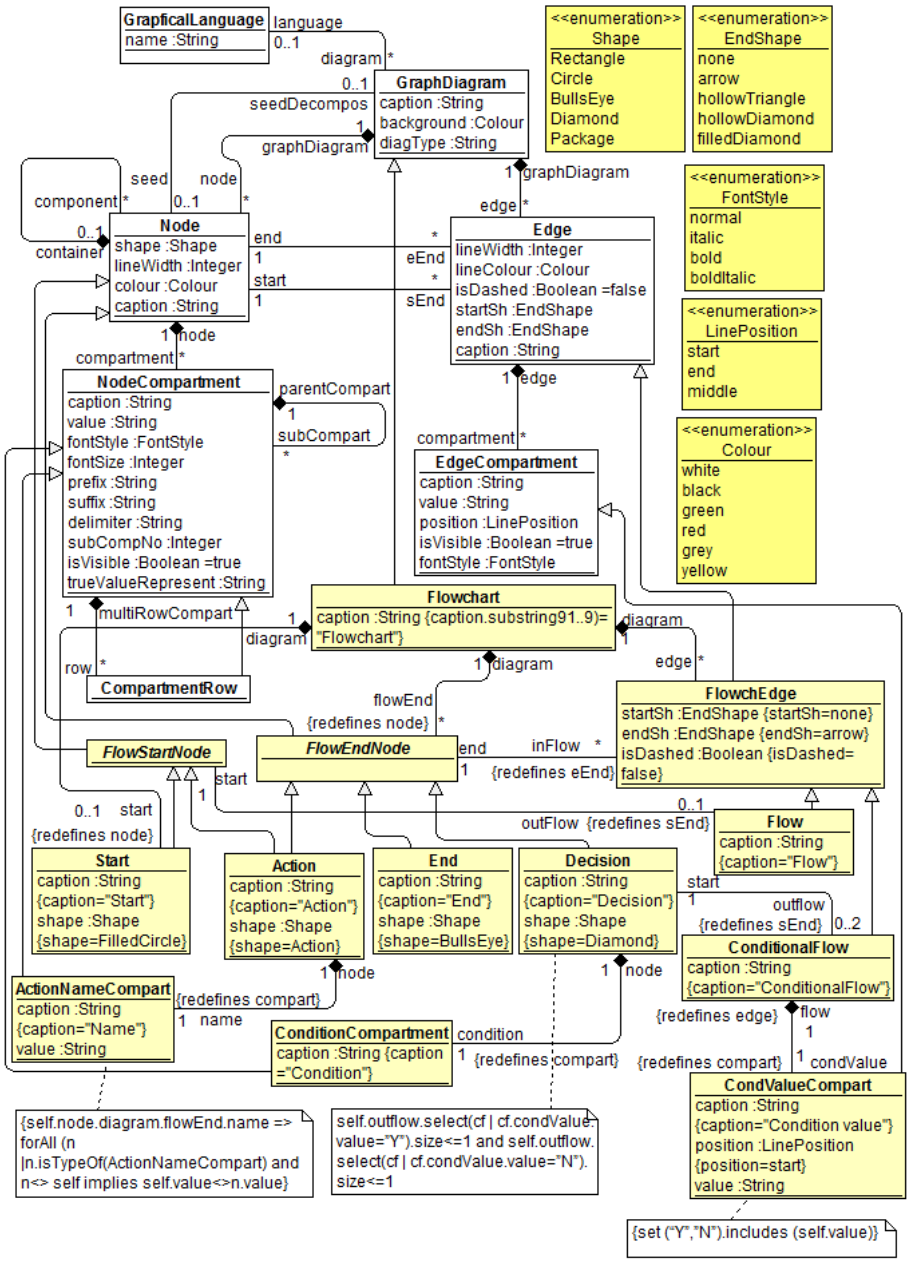

Fig. 5. Flowchart diagram syntax definition by specialization

In fact only the specialized classes matter, all relevant attributes and associations are in this part.

We see that the specialized metamodel for flowcharts in Fig. 5 is quite similar to the "naïve" flowchart metamodel in Fig. 2. Both metamodels have the same classes for node and edge kinds in a flowchart, only in Fig. 5 they are subclasses of UMM classes. Abstract superclasses are used with the same goal to simplify the usage of UML multiplicities for a precise specification of node-edge relations – which edges can start from which nodes. The main difference from Fig. 2 is that diagram element attributes are replaced by the relevant compartment classes – the texts for nodes and edges are important syntax elements as well. The attributes in Fig. 5 have a completely different role, they specify some of the element style features which are significant for the syntax definition. The values of these attributes are fixed via the OCL constraints attached to them. Attributes from UMM not used here are not repeated in the specialization (but formally they are present). The same OCL constraints expressing essential non-local restrictions on flowchart elements are attached to three classes. Their form is slightly different because attributes are replaced by compartment classes. It should be noted that they are built completely in terms of the specialization classes and associations. Thus the principle is supported that any syntactically valid flowchart is a direct instance of the specialized metamodel.

Since all the information relevant for the metamodel reader is contained in the specialization, we propose a custom notation for metamodel specialization in order to make metamodel diagrams more compact. The custom notation for the same flowchart example is shown in Fig. 6. There only the specialized classes are shown, with the UMM class name (of which it is a subclass) shown in braces and in bold italic font. The association end redefinition is shown without the *redefines* keyword, with the original name from UMM in a similar style.

The semantics of standard UML class specialization certainly is retained completely. For example, when an abstract class in the specialization has the name of the corresponding UMM superclass inserted, it extends to all of its subclasses in the specialization as well. The given custom notation will be used also for all the next specialization examples.

Definitely, the flowchart diagram example is not a very complicated one. To see how the approach applies to more realistic diagram definition, look at the EMOF level class diagram editor definition fragment (Fig. 9) in Section 4. For syntax definition there only the input control and palette related attributes should be removed (and some OCL constraints modified). The complete syntax definition for such a class diagram would require two more diagrams of a similar size.

We conclude this section by some comments on the current status of graphical language syntax definition. The precise graphical syntax definition of languages has been slightly neglected for a while. For example, the graphical syntax of UML up to the version 2.4 (OMG, 2011) has been defined as informal comments and example pictures in the UML documentation. Only starting from UML 2.5 (OMG, 2015 a), some formalization is offered also for defining the graphical syntax of diagrams. It is based on

the new OMG standard for Diagram Definition (DD) (OMG, 2015 b). However, the main goal of this standard is to enable diagram interchange (DI) between modeling tools implementing the same language, but not a simple and precise diagram syntax specification for tool developers. Therefore DD consists of Diagram Interchange (DI)

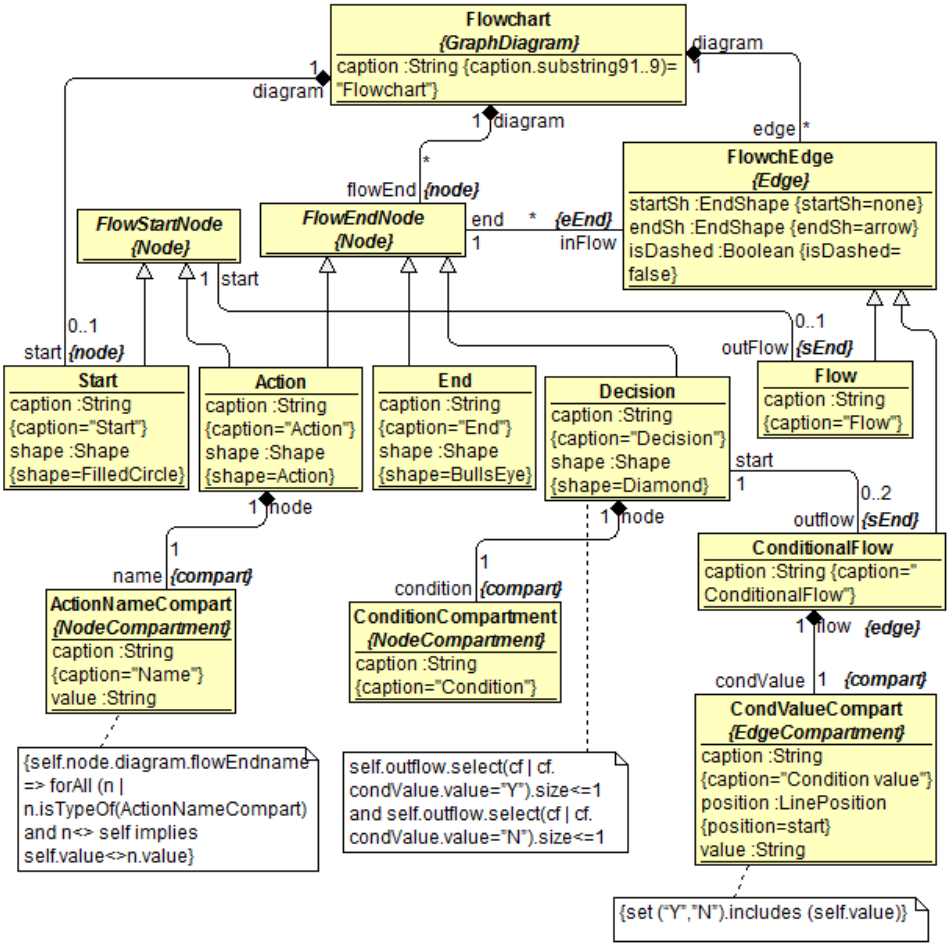

Fig. 6. Flowchart syntax definition in custom notation

and Diagram Graphics (DG) parts, each having a metamodel. The DI part permits to describe the logical structure of a diagram at a very high abstraction level. The second component of DD is the DG (Diagram Graphics) metamodel, which is oriented towards a low level specification of graphical element rendering in a tool. The complete concrete syntax specification for a diagram therefore requires also a transformation definition for mapping DI elements to DG elements – standard graphic elements (shapes, lines and text elements). To have a metamodel which describes the diagrams in the given language by more understandable concepts it is proposed to specialize the DI metamodel to an

interchange metamodel for the language. Thus, for UML diagrams the specialized UMLDI metamodel is given as an example in the DD standard (OMG, 2015 b). It should be noted that the UMLDI metamodel is obtained from DI metamodel by a specialization approach quite similar to that used in this paper, yet more complicated specialization features here have to be used. Due to all these complexities in the DD approach, the main description of UML diagram syntax is still given informally in the UML 2.5 standard (OMG, 2015 a). There are also attempts to specify the graphical syntax by means of specific graph grammars (Rekkers and Schurr, 1997, Costagliola et al, 2004), but they also do not offer a simple solution because the application of rules there is much more complicated than for textual grammars. Thus the problem of a simple, but at the same time precise definition of the graphical syntax for a graphical modeling language is still open.

4. Graphical Diagram Editor Definition by Metamodel specialization

As it was already stated the diagram syntax definition for a graphical language can be easily extended to a graphical editor definition for this language. The complete version of the universal metamodel from Fig. 4 must be used now – with attributes displayed in bold also included. These attributes specify the basic properties of user interaction with the editor – the palette structure and style for diagram creation and input controls for entering the corresponding compartment values.

However, the usage of these attributes becomes clear only in the context of the new concept for editor definition – the Universal Engine (UE). Universal engine is an abstract editor whose generic behavior is explained in terms of UMM. It can work with instances of UMM classes such as GraphDiagram, Node, Edge etc. and perform typical editing actions related to them. The details of such actions are determined by attribute values of these classes. Most of these attributes must have fixed values set for a language editor, and these values are set only in the specialized metamodel for the given language. Therefore the UE behavior is precisely defined only by a UMM specialization.

In a more practical setting there is one more extension of UMM for editor definition in our approach. Typically any real diagram editor contains the concept of Project – a set of related diagrams having a common usage. The contents of a project has to be somehow visualized – frequently via a tree. However, since we want to restrict our visualization facilities, a Project diagram is introduced instead. It contains Diagram seeds – nodes from which the corresponding diagram can be accessed via double-click. Thus a project diagram is a normal graph diagram (containing only nodes). To represent this extension we add one more version of UMM for editors – see Fig. 7, where the only added element is the Project class. As before, there the editor-related elements are in bold style.

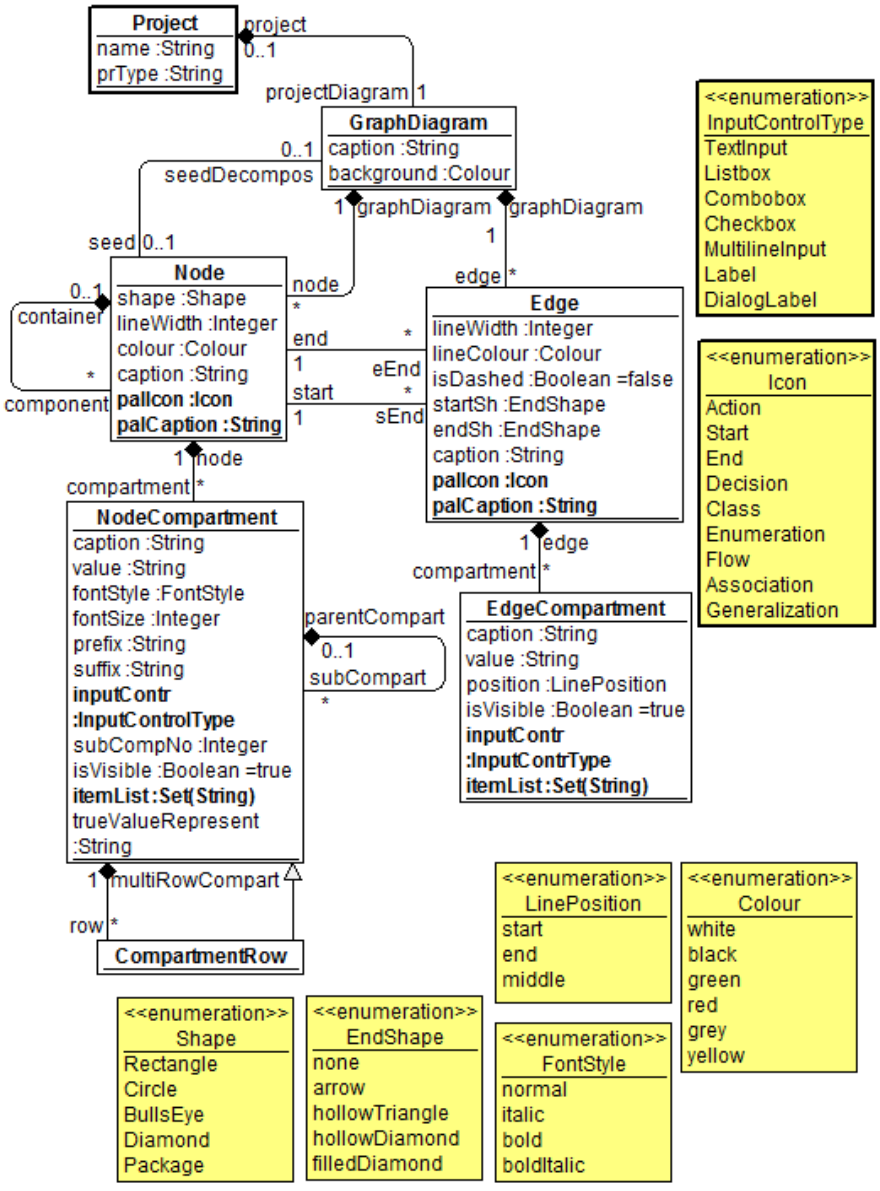

Fig. 7. The practical extension of UMM for editor definition

Now we can provide a list of typical actions supported by the UE for editors:

- Manage the current project, in particular, add new diagrams to it of the kinds defined by the available specializations (using the palette of the project diagram)
- When a diagram is opened (existing or new), its palette is opened as well. The palette for a diagram kind is generated by UE on the basis of its specialization (diagram element types having the related palette style attributes set)
- A new element in the diagram is created when the user clicks on the relevant palette element and selects its position in the diagram (or end nodes for a new edge)
- If the element contains text compartments in its specialization, the corresponding dialog form is opened by UE, the form contents is defined via the corresponding input control attributes in the specialization (and the subcompartment structure)
- Each subcompartment has its own input control included in the form for parent compartment, for multi-row compartments the value entry row-by-row is supported
- Compartments of an existing element may be opened for modification
- UE supports a number of standard actions not dependent on a specialization – creating, saving and opening a project, modifying a diagram layout, modifying a diagram element style, copying a diagram element etc.
- UE checks the applicability of user actions on the basis of the diagram syntax defined in the specialization and checks the validity of user input on the basis of provided OCL constraints

Further details of the behavior of UE will be explained on editor definition examples. There is one general note on the used UMM specialization style. While for syntax definition the values of attributes were fixed by OCL constraints attached to the attributes, for editor definition it is more natural to define the values simply as default values in UML. This is because many of these values are used as defaults just at the moment of creation of a new element (e.g. node style attributes), later on they can be modified by the user via the supported "auxiliary" functionality of UE (UE simply doesn't permit to modify the strictly fixed attributes). The default value may be specified by a constant or a true OCL expression, but also OCL constraints may be used for non-modifiable values. The custom notation – only the specialization – will be used for both examples. Certainly, when defining an editor, we must think that the editor works on instances of the specialized metamodel and the user creates instances of specialized diagrams kept in a specialized project as a repository. This doesn't contradict to the principles of UE presented before because any instance of a specialized class can be treated also as an instance of the corresponding superclass from UMM (this view is used in Section 5 for implementing the editor platform based on the specialization approach).

Fig. 8 shows the editor definition for flowchart diagrams according to the syntax defined in the previous section.

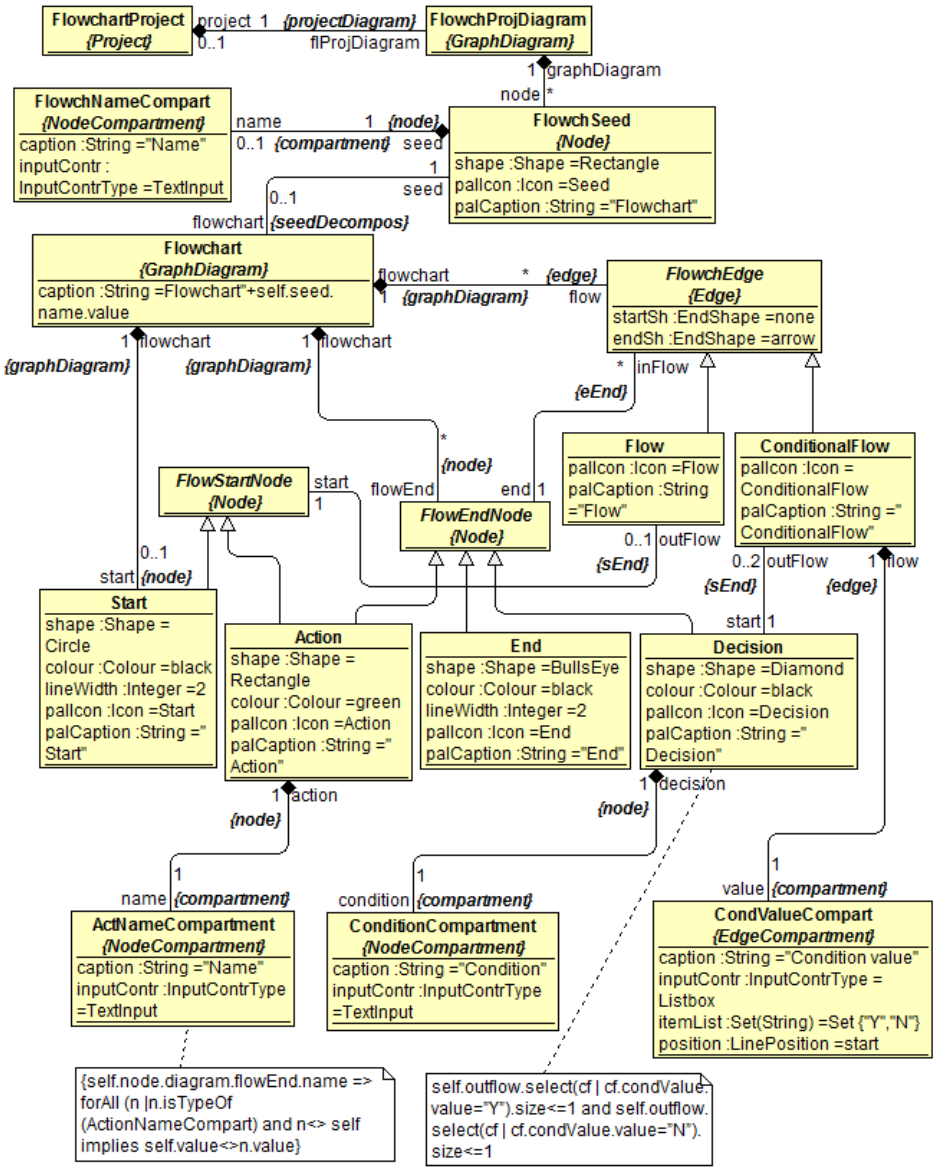

Fig. 8. Flowchart editor definition

The editor definition contains the same specialized classes as the flowchart syntax definition in Fig. 5 and 6. Now more attributes have values set – here using the default value feature. First, some comments on the Project definition. Our language contains just one diagram type, therefore the specialized FlowchProjDiagram contains only one seed node type – FlowchSeed. Since the palette attributes are specified for this node kind, the project diagram will show a generated palette with just one element – the flowchart seed.

When the user clicks on this element, a new seed node is created and the simplest form consisting of just one text field and the caption Name is shown. When the user there enters a string, this value is used both as the seed and diagram name (see how the caption value for a flowchart is set). The new flowchart diagram appears when the user double-clicks the seed.

A flowchart diagram (new or existing) appears with a generated palette containing four node elements and two edge elements – for all specialized non-abstract node or edge classes – they all have the palette related attribute values set. Abstract classes in the specialization are not used for palette element creation – they are not uniquely linked to one specialized node or edge class. To create a node in the diagram the user has to click the corresponding palette node and then click on a free place in the diagram canvas. The node is created in this position – UE can automatically move the existing diagram elements to find enough space. Before the new node appears, UE opens a generated dialog form, if the node class owns at least one compartment. If there are several compartments attached, they appear in the same form (in the order specified by subCompartNo attribute value). If the compartment has subcompartments, a subform is shown when the user selects this compartment (see this feature in detail in the next example). The flowchart editor has compartments defined only for Action and Decision nodes. Each of them has only one compartment to be entered via the simplest input kind – the text input, therefore a dialog form with one text field is shown for these nodes.

To create an edge (here a flow or conditional flow), the user has to click the corresponding palette element and then select an existing start node and end node in the diagram. If there is a compartment attached, the relevant dialog form is opened. In our example the conditional flow has one compartment – for entering the condition value. The specified input control is a listbox, where the user can select a value only from the displayed item list. The itemList attribute defines the values in this list, here it is a fixed list containing two values – Y and N. Since the user can select only one of these values, there no more a need for an OCL constraint on the compartment value, as it was specified in the syntax definition in Fig. 5. Certainly, OCL constraints still may be needed for more complicated situations – the name compartment for the Action node has the same constraint as in the syntax definition. But in editor definition it has a new semantics – the editor evaluates this constraint when the user has completed the value input. If it evaluates to *false*, a standard error message on invalid value is displayed. Similarly, the second OCL constraint related to condition values is a complex one involving the values on all edges outgoing from a decision node. This constraint has to be retained from the syntax definition as it is. Now it reports as an error, e.g. the situation when the user tries to enter the value "Y" also for the second edge exiting the given node. The edge multiplicity constraints in plain UML notation are also treated this way, e.g. an error message is shown when the user tries to start the second Flow from an Action node or he/she tries to place the second Start node in a diagram. Thus only correct flowchart diagrams according to the syntax definition can be built. Certainly, the editor designer has to take into account the way how static constraints in syntax

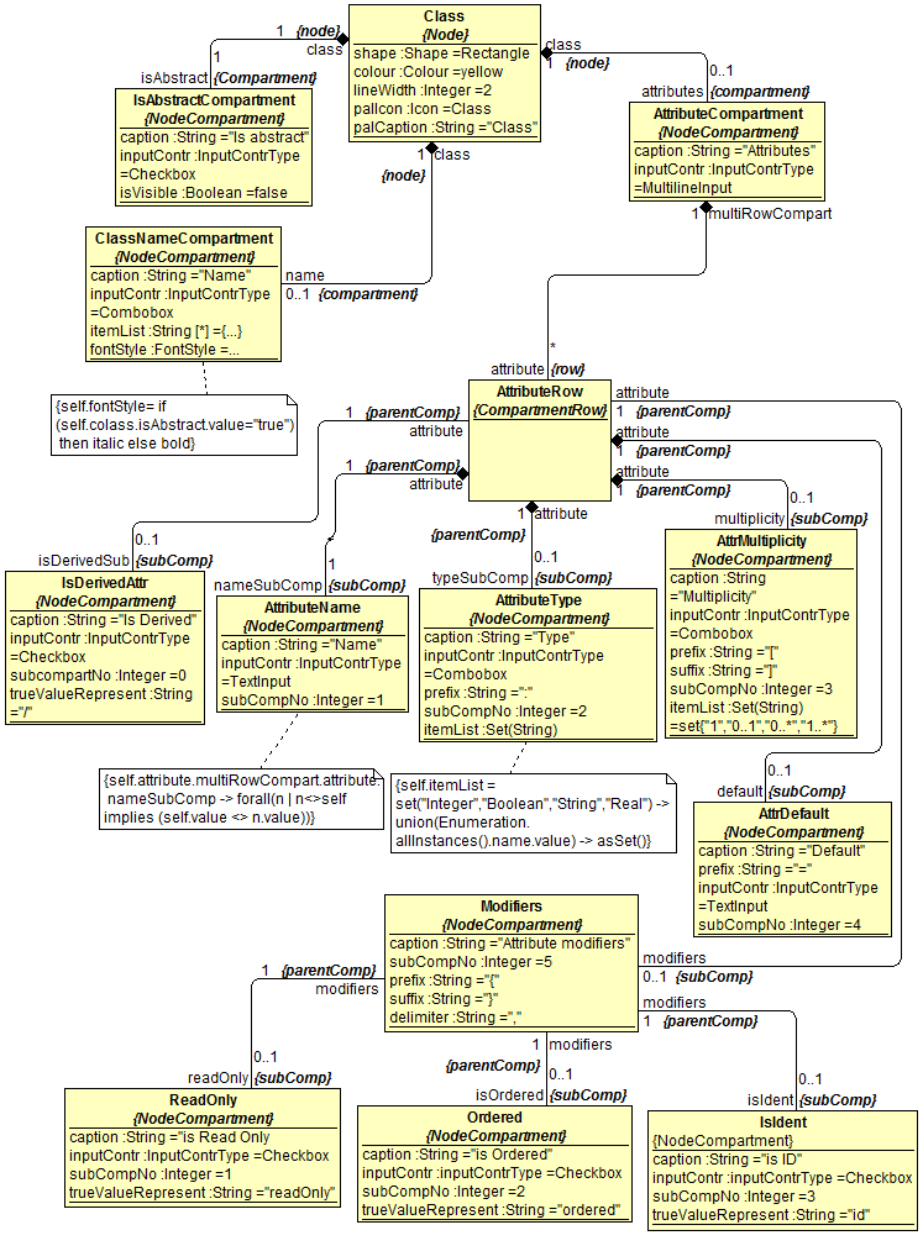

Fig. 9. The attribute fragment of EMOF class editor definition

definition are interpreted as active constraints in the editor, thus adaptations of OCL constraints may be sometimes needed.

Now the second editor example – a class diagram editor for the subset of UML which corresponds to the EMOF metamodel (OMG, 2015 c). Fig. 9 shows an editor fragment – the features related to Attribute in a Class – the multiline Attribute compartment and all details of an Attribute according to EMOF specification. The complete editor definition has to support the Class and Enumeration nodes and Association and Generalization edges. The Class node in its turn includes all features for EMOF support. To define all this, two more diagrams of similar size are required.

The main goal of this example is to illustrate how node compartments with a complicated structure are defined in our approach. This refers both to graphical syntax definition and editor definition. The example in Fig. 9 shows the complete definition of the Attribute compartment in a Class node. It is a typical multiline compartment where each line has a complicated substructure. In addition, the Class name and isAbstract compartments are also included – to illustrate the specific notation for showing that a class is abstract. A more elaborated use of OCL constraints in the context of editor definition is also shown.

The metamodel attributes which have to be set to a constant value are defined via the default value option, but those whose value must be set via a proper OCL expression are defined by OCL constraints, especially when this value is dependent on other diagram elements and may change over time.

The multiline nature of the AttributeCompartment is defined by the choice of the input control – MultilineInput. This control displays all existing lines in a multiline field, with a possibility to add a new line or delete an existing one. Each line there represents a separate class attribute, the editing of such line is enabled by the AttributeRow class in the specialization (which specializes the CompartmentRow in UMM – a feature directly introduced for such situations). Since such row is also a Compartment, it can have a substructure defined via the specializations of subComp association (here six subcompartments are present according to the attribute syntax in UML). The editor (in fact, UE) supports this feature by opening a new dialog form for editing an existing row or entering a new one. This form contains a field for each subcompartment, their order is defined via the subCompNo attribute. The visible text value of the line is obtained by concatenating all these values. The IsDerivedAttr subcompartment uses the checkbox for entering the Boolean value, but entering the *true* value results in creating the string "/" (according to the attribute trueValueRepresent setting) – and *false* results in nothing. Several subcompartments have the values of prefix or suffix attributes set, these string values are prefixed (or suffixed) during the concatenation, e.g. prefix ":" for the attribute type. If the subcompartment value is empty, prefixes or suffixes are ignored. For attribute type and multiplicity input the Combobox input control is used. Like Listbox, it presents an item list to select from, but permits also simple text input for entering a different value. The item list is specified via the itemList attribute. It is set to a fixed string set for multiplicity. But it uses an OCL constraint for the type. This is because the item list can dynamically extend, when the user adds instances of other diagram elements (here instances of Enumeration – not shown in the fragment in Fig. 9). One more situation is demonstrated for the Modifiers subcompartment – it itself has a nested

substructure, with a number of features present or absent. Therefore each feature is entered via checkbox and has a string for representing *true*, but the delimiter attribute specifies how the strings should be separated during the concatenation, if more than one is present. The complete string value of Modifiers is included in braces using the prefix and suffix.

The OCL constraint for attribute name is a typical input value check, producing an error message when a duplicated attribute name is entered for a class.

Now some comments on the Class name compartment. According to the UML specification the fact whether a class is abstract is visualized by the font style of the class name. The user can enter the isAbstractCompartment value via the standard checkbox – the UE internally stores the entered value as strings true or false. But this compartment itself must not be visualized, therefore the attribute isVisible is set to false in this subclass. Instead, the ClassNameCompartment style must be set to italic if the class is abstract and to bold if it is not. Exactly this fact is specified by the OCL constraint attached to ClassNameCompartment (the definition by a constraint ensures also that the given relationship remains in force when the user modifies the isAbstract value).

The provided examples confirm the fact that a typical diagram editor functionality can be defined this way. Certainly, advanced value prompting and value checks present in commercial UML editors would require significantly more complicated OCL constraints, frequently related to the abstract syntax of the language as well. However, our approach is mainly oriented towards such graphical DSML support where typically only features similar to those shown here are required.

5. Editor Workbench implementation Principles

In this section we briefly provide the basic principle how an editor definition workbench based on metamodel specialization could be implemented. This implementation complies with the main principle of the approach that the Universal engine (UE) is based on the original UMM. The proposed practical Universal engine runtime will consist of several parts. In a sense, this architecture will be similar to the existing TDA platform structure (Sprogis, 2010, 2013), from which some parts could be more or less directly reused. A new component is the Specialization Engine (SE), which is the sole component managing the current model (project) processed by the editor according to the specialized metamodel. In addition, it provides a view (in fact, a copy) of model fragments (typically, the current diagram being edited) to be used by other parts of UE according to UMM. These other parts are the Graph Diagram Engine (or Presentation Engine) – GDE, the Dialog Engine (DE) and the Main Engine (ME).

The Graph Diagram Engine performs all “technical” work in rendering diagrams – simply displays a diagram, adds a new element with a known content to it and updates the layout accordingly, modifies the layout upon a user request etc. It also accepts the user actions in a diagram area and, if the action is a “logical” (not purely technical one), creates a notification (event) for other UE components to react upon. The main value of this engine is a nontrivial algorithm for creating readable diagram layouts. The Dialog Engine performs all the technical work in displaying the generated dialog forms for compartments and reacting to user inputs in form fields – but it doesn’t perform logical

processing of the input values. The Main engine in the existing TDA (in fact, a universal interpreter) performs all logical actions on the basis of the given Type metamodel instance. The Main engine in the existing TDA is very complicated and contains a large functionality not relevant for the specialization approach, because in the existing version many low level actions can be configured or even custom functionality can be added to them (via custom model transformation procedures). All these actions are supposed to have a standard behavior in this approach.

The Graph Diagram Engine (or Presentation Engine) – GDE and Dialog Engine (DE) could be reused nearly completely from their versions in TDA. The Main Engine also could be reused to a significant level, but with some code modifications (certainly, only a part of it is really needed here). This is based on the fact that UMM is very close to the Type metamodel (in fact, a subset) in TDA – only the coding differs in some places.

The diagram set built using a defined editor is stored in a Project repository (PR) according to the specialized MM (SMM) – as a normal instance set. This repository is being serviced only by SE, no direct diagram editing occurs there. The diagram editing occurs (by UE components) only in a Temporary repository (TR) which contains instances according to UMM. Normally only the current diagram must be kept there (sometimes two diagrams are required). SE supports the copying of a diagram from PR (according to SMM) to TR (according to UMM). In fact, this is a simple operation, since an instance of an SMM class is also an instance of the corresponding UMM class, and attributes are the same (not redefined, only values are set). Only the links of redefined associations have to be stored as links of the original associations.

In addition, when UE modifies an instance in TR or creates a new instance, it has to notify SE about the modifications (reference is by a technical Guid attribute maintained by SE and UE), then SE synchronizes the elements in PR.

The OCL expression evaluation is done by SE only in PR (upon requests by UE).

The UE behavior is fixed to a standard schema, no explicit configuration is planned. For a diagram new elements are created according to the supplied palette for this diagram kind. Node or edge compartment editing also occurs according to the supplied compartment dialog schema for the element. Pop-up menus also are fixed.

The palettes and dialog schemas for elements are created from specializations. There will be a language developer mode in the workbench, where one or more UMM specializations will be built, in totality constituting a graphical language. When the specialization is complete, it is “compiled” to a palette tree, which contains the palette for the project diagram – in its turn containing palette seeds for all defined diagram types in the language. Under each project palette seed a local palette tree for the corresponding diagram type is stored. This tree contains the diagram palette with an element for each specialized node or edge. Under each such element the complete compartment subtree for this element (as defined in the specialization) is built. Such a subtree related to a diagram element is called the element template. For all attributes with default values set in the specialization these values are present in the template. All palette tree elements are instances of the palette tree metamodel classes – an instance for each SMM element –

node, edge or compartment. Fig. 10 shows the Palette tree metamodel and its relations to the UMM for editor definition. Some purely "technical" runtime-related attributes are added to UMM classes there as well.

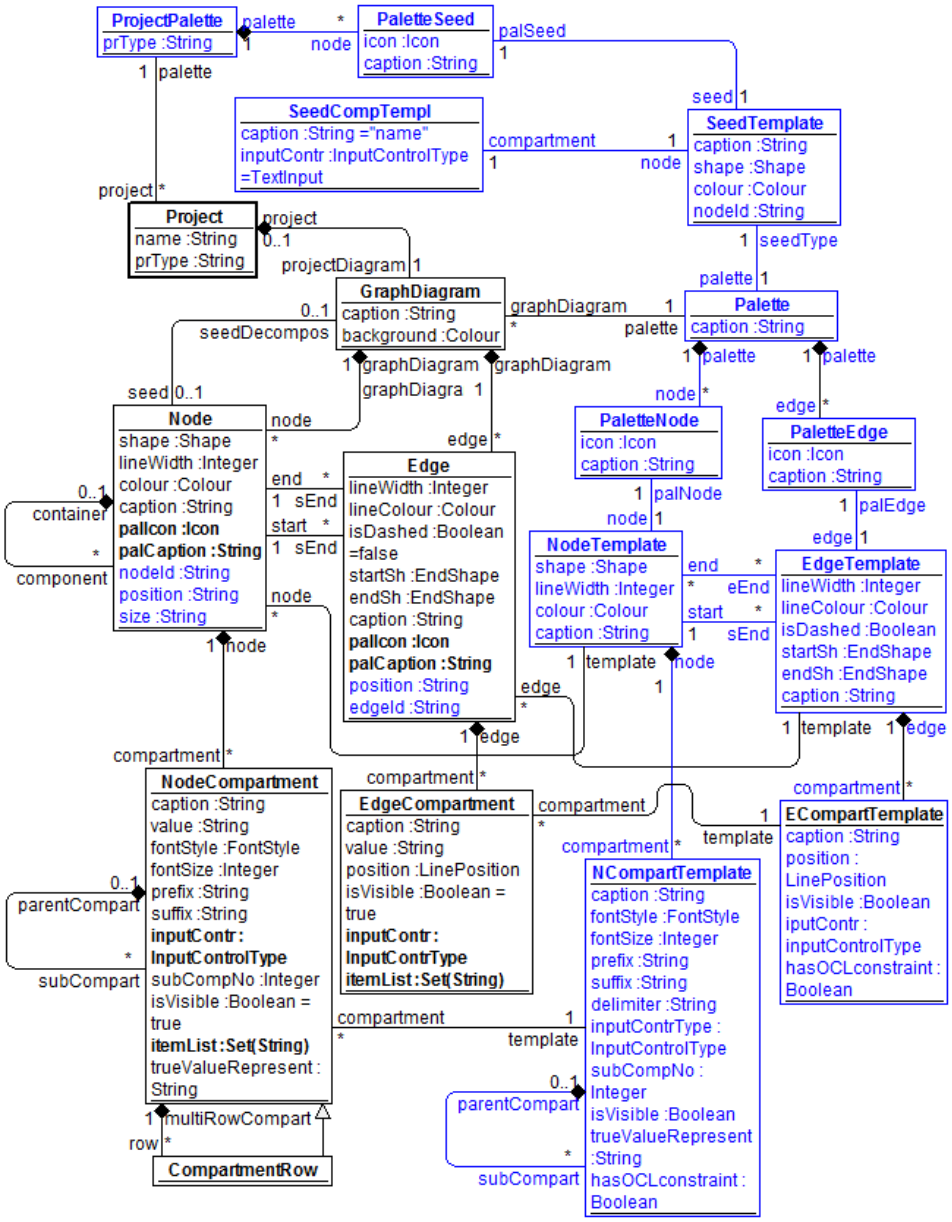

Fig. 10. The "practical" UMM version together with Palette tree metamodel.

Certainly, a more efficient implementation of the modified Main engine may require some modifications of the palette tree metamodel, but the general idea will be preserved.

Thus the components of UE have to build a true diagram element from the corresponding template by setting the remaining "dynamic" attribute values according to the editor user wishes. Fig. 11 shows a fragment of the Palette tree with element templates for the flowchart editor in Section 4. This fragment is shown as an object diagram in UML.

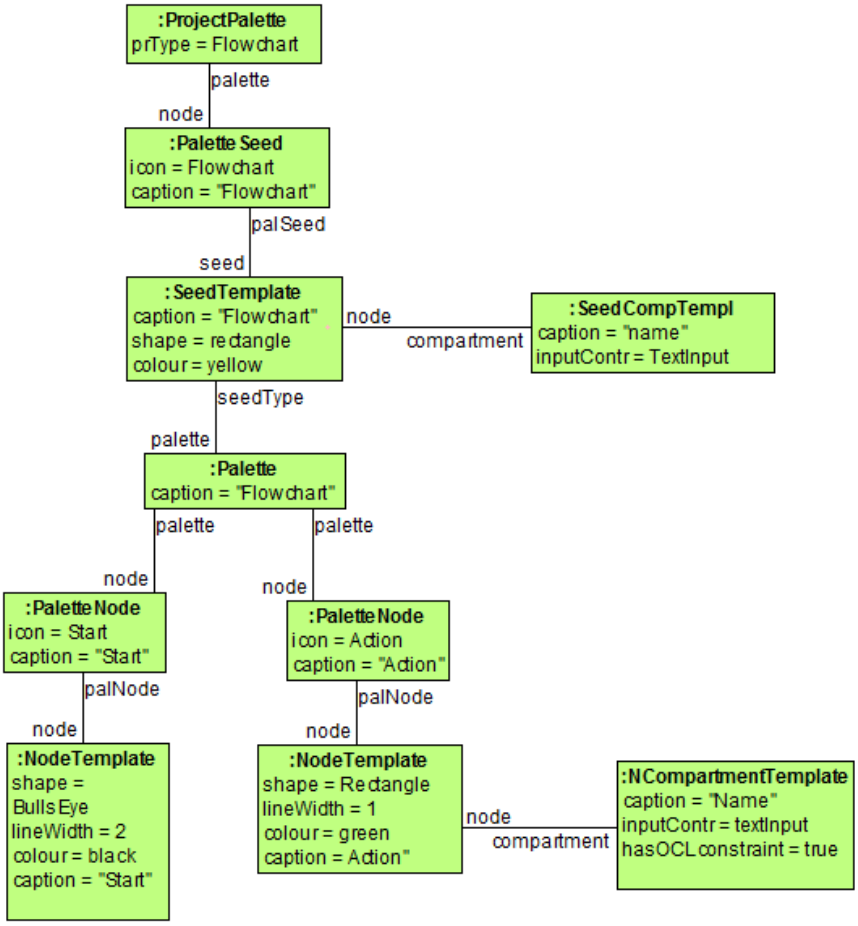

Fig. 11. Fragment of the Palette tree for Flowchart editor

The proposed implementation schema for a specialization based editor workbench thus is expected to require not a very large effort since several components of UE could be reused from the existing TDA. The only completely new component would be the specialization engine for synchronizing the model in PR with its parts in TR and performing some general management. The language developer workplace could be implemented on the basis of an appropriate open-source UML tool in Eclipse, e.g. Papyrus (WEB, f). Then one the existing OCL interpreters, e.g. Eclipse OCL (WEB, g) could be used as well. All this is possible since metamodel specialization is completely based on standard UML features. The specialization compiler to the Palette tree format could be built as an Eclipse plugin, most probably in one of the model transformation languages available there.

Conclusions

The paper proposes a further development of ideas presented previously by authors on the usage of metamodel specialization for the definition of graphical languages and their support tools. The approach has been explained in a more detailed way, especially the Universal metamodel and Universal engine. The provided examples of language and editor definition show that the specifications obtained by metamodel specialization are precise, complete and sufficiently simple at the same time, and they use only standard features of UML class diagram. A more detailed insight into the editor workbench implementation has been provided as well. It contains sufficient details to estimate the required implementation effort, which is not large. We see that the main application area for the approach is graphical DSL support, and the popularity of graphical DSL is growing significantly. We plan to approve the approach in practice by implementing at IMCS UL a new specialization-based platform for graphical DSL support, using the ideas presented here. And we see that in perspective the approach could be applied to a much broader class of tasks.

Acknowledgements

This work is supported by the Latvian National research program SOPHIS under grant agreement Nr.10-4/VPP-4/11

References

Barzdins J. et al. (2007). GrTP: Transformation Based Graphical Tool Building Platform. In: *Proc. of MDDAUI'07 Workshop of MODELS 2007*, Nashville, Tennessee, USA, CEUR Workshop Proceedings, volume 297, 4 pp.

Barzdins J., Rencis E., Kozlovics S. (2008) The Transformation-Driven Architecture. In *Proceedings of DSM'08 Workshop of OOPSLA 2008*, Nashville, Tennessee, University of Alabama at Birmingham, 60 – 63.

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

Costagliola G., Deufemia V., Polese G. (2004). A Framework for Modeling and Implementing Visual Notations with Applications to Software Engineering. *ACM Trans. Softw. Eng. Methodol*., 13(4): 431–487.

Grune D. et al. (2012). *Modern Compiler Design*. Springer, New York.

Juliot E., Benois J. (2009) Viewpoints creation using Obeo Designer or how to build Eclipse DSM without being an expert developer? Obeo Whitepaper. http://spotidoc.com/doc/197222/

Kalnins A., Barzdins J., (2016 a) Metamodel Specialization for DSL Tool Building, In *Databases and Information Systems, DB&IS 2016 Proceedings*, CCIS 615, Springer, 68-82.

Kalnins A., Barzdins J., (2016 b) Metamodel Specialization for Graphical Modeling Language Support, In: *Proceedings of MODELS 2016, 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ACM, 103-112.

Kalnins A., Barzdins J., (2016 c). Metamodel Specialization for Diagram Editor Building, In *Databases and Information Systems IX, Selected Papers from DB&IS 2016, Frontiers in Artificial Intelligence and Applications*, Vol. 291, IOS Press, 87-100.

Kelly S., Tolvanen (2008), *Domain-Specific Modeling: Enabling Full Code Generation*, John Wiley & Sons, Hoboken, New Jersey.

OMG (2011). Unified Modeling Language (UML) – Version 2.4.1 – OMG document formal/2011-08-05.

OMG (2012). Object Constraint Language (OCL) – Version 2.3.1 – OMG document formal/2012-05-09.

OMG (2015 a). Unified Modeling Language (UML) – Version 2.5 – OMG document formal/2015-03-01.

OMG (2015 b). Diagram Definition (DD) – Version 1.1 – OMG document formal/2015-06-01.

OMG (2015 c). Meta Object Facility (MOF) Core Specification – Version 2.5 – OMG document formal/2015-06-05.

Rekkers J., Schurr A. (1997). Defining and Parsing Visual Languages with Layered Graph Grammars, *J. Vis. Lang. Comput*., 8(1), 27–55.

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

Sprogis A. (2013). *Configuration Language for Domain Specific Tools and its Implementation*. PhD thesis (in Latvian), University of Latvia, Riga.

WEB (a). Graphical Modeling Framework (GMF, Eclipse Modeling subproject), http:// www.eclipse.org/modeling/graphical.php.

WEB (b). Eclipse Modeling Framework (EMF). https://projects. eclipse.org/projects/ modeling.emf/

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

WEB (d). Sirius Overview. http://www.eclipse.org/sirius/overview.html

WEB (e). EuGENia Live. http://eugenialive.herokuapp.com/.

WEB (f). Papyrus Project in Eclipse. http://projects.eclipse.org/projects/modeling.mdt.papyrus.

WEB (g). Eclipse OCL (Object Constraint Language). https://projects.eclipse.org/projects/ modeling.mdt.ocl.

Received November 30, 2016, accepted December 3, 2016