Databases and Information Systems IX
G. Arnicans et al. (Eds.)

doi:10.3233/978-1-61499-714-6-87

Metamodel Specialization for Diagram Editor Building

Audris KALNINS[1] and Janis BARZDINS
Institute of Mathematics and Computer Science, University of Latvia, Riga, Latvia

Abstract. Domain-specific diagram editor building environments nowadays as a rule involve some usage of metamodels. However normally the metamodel alone is not sufficient to define an editor. Frequently the metamodel just defines the abstract syntax of the domain, mappings or transformations are required to define the editor functionality for diagram building. Another approach [8] is based on a fixed type metamodel describing the possible diagram elements, there an editor definition consists of an instance of this metamodel to be executed by an engine. However there typically a number of functionality extensions in a transformation language is required. The paper offers a new approach based on metamodel specialization – by just creating subclasses. First the permitted metamodel specialization based on standard UML class diagrams and OCL is precisely defined. A universal metamodel and an associated universal engine for the diagram editor domain is described, then it is shown how a specific editor definition can be obtained by specializing this metamodel. Examples of a flowchart editor and UML class diagram editor are given.

Keywords. Metamodeling, metamodel specialization, DSL tools, diagram editors

1. Introduction

Metamodeling typically is the basis for most domain-specific diagram editor definition platforms nowadays, but the principles how metamodels are used vary significantly. Many such tool building platforms are related to Eclipse EMF and GMF frameworks [1,2]. They all are oriented towards the "classical" diagram building approach where at first a domain metamodel describing the abstract syntax of the language must be defined in EMF, only then the graphical concrete syntax (presentation metamodel) is described as a GEF metamodel, with a mapping metamodel between the both added (in GMF). There are some improvements of the basic Eclipse approach such as ObeoDesigner [3] where the presentation metamodel can be defined as a viewpoint of the domain metamodel, or Eugenia [4] where the presentation and mapping metamodels are defined as annotations to the domain metamodel and then generated using a transformation language. Thus there the basic Eclipse pattern – start with the domain metamodel is preserved. A completely different platform – Microsoft DSL [5] uses a similar pattern by starting with a domain metamodel and then adding the presentation and mapping metamodels, only metamodels are created in a "dialect" of UML. A completely domain specific metamodeling language GOPRR is used in the MetaEdit [6] platform where the graphical syntax metamodel can be defined directly but with a limited

[1] Corresponding Author, e-mail: audris.kalnins@lumii.lv

functionality. A common feature to all these platforms and some similar ones is that for each new DSL a new metamodel must be created in some metamodeling language.
The platform devoted most directly to graphical modeling language editor definition is the platform developed by IMCS UL – TDA [7,8] (the platform initially was named GrTP [9]). There a fixed Tool definition metamodel is proposed which defines type classes for all DSL elements – GraphDiagram, Node, Edge, Compartment, Palette etc., in addition style classes for all these elements are also present there. Then a concrete DSL and an editor for it is defined as an instance set of this metamodel. However the complete metamodel for an editor contains also the runtime elements – the classes GraphDiagram, Node, Edge etc., thus instances of completely different nature – Node, NodeType and NodeStyle (and so on) coexist in a runtime model corresponding to this metamodel. In addition, these instances of semantically different layers must be properly linked. The approach is quite usable for simple DSLs where the type instance set defining the language can be created by an auxiliary tool – the Configurator [10], without deep knowledge of the metamodel. However even for slightly more complicated languages the mechanism of extension points related to events of relatively low abstraction level and associated custom transformations has to be used. To create such a transformation (in a special Lua/lQuery language [11]) the developer has to have a deep knowledge of the metamodel. The editor runtime is based on a Universal Interpreter – a type metamodel interpreter which interacts with the custom transformations and the support engines for managing diagram layout and user dialogs (Presentation Engine and Dialog Engine).

During the TDA development attempts have been made to use also alternative styles of metamodel usage, closer to the topic of this paper – metamodel specialization. One such attempt [8,12] was to use an extended UML stereotype mechanism and a stereotype specialization. Another one [13] was the extension of UML class specialization by non-standard concepts borrowed from OWL. However none of these ideas are based on a clean UML usage and none of them were completed and implemented.

This paper proposes a completely new approach to editor definition for Domain Specific Modeling Languages (DSML) on the basis of their graphical syntax. The approach is based on a consistent use of metamodel specialization. In Section 2 the metamodel specialization based solely on standard UML class diagram elements and OCL is precisely defined. It should be noted that though the concept of subclass is widely used in metamodel building the whole metamodel specialization to a new more detailed metamodel is a new idea in metamodeling. The only reference where the term "metamodel specialization" has been explicitly used is [14] where the concept has been used for DSML extension, but within a significantly different context. In addition, this idea (without explicitly naming it) has been used in the OMG standard for Diagram Definition [15]. Section 3 introduces the concepts of Universal Metamodel (UMM) for the graphical diagram domain and a Universal Engine (UE) related to this metamodel. The concepts are explained on a very simple flowchart editor example. Section 4 describes a complete UMM for DSML definition and the main functionality of UE providing a realistic editor behavior. Section 5 gives a complete specialization of UMM for a realistic flowchart editor. Section 6 presents some most interesting fragments of a class diagram editor defined by a specialization, there the proposed approach for defining the internal structure of texts in a diagram is illustrated as well. Both these examples confirm that DSML definition by specialization is the cleanest and most understandable way. Finally, the basic principles of an implementation of the approach are given in

Section 7. This paper is an updated version of paper [16] presented at DB&IS'2016, the figures 1 to 5 are from this paper.

2. Metamodel Specialization

Class specialization 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. However, there is a variation of specialization which can provide a completely new idea in building class models. It is the specialization of a whole metamodel.

A widely used UML concept is an abstract class, which cannot have instances directly. To be more precise, here we need this concept in two contexts. The first one is the standard usage of an abstract class in UML models, where it is being defined as a union of instance sets of all its subclasses. This kind of usage of an abstract class will appear here for technical purposes to pull up common properties for several subclasses.

The other kind of usage of abstract classes appears when we specialize the whole metamodel. Classes of such generic (base) metamodel are also abstract in the sense that they do not have direct instances in our approach. However, there is no need to define their instance sets somehow – only their subclasses appearing in a metamodel specialization will be really used. The classes of the generic metamodel will be used to assign an intuitive semantics to a set of classes and their properties, with more details appearing only in a specialization.

Now let us give an example of a generic metamodel in Figure 1.

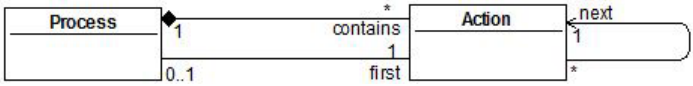

Figure 1. Example of a generic metamodel.

This very simple class diagram containing just two classes still gives some guidelines of its intended meaning – it represents a simple kind of Process metamodel containing a sequence of actions. Now let us create a specialization of this metamodel by creating subclasses of classes in Figure 1. This specialization presents a simplest kind of Business Trip process – see Figure 2.

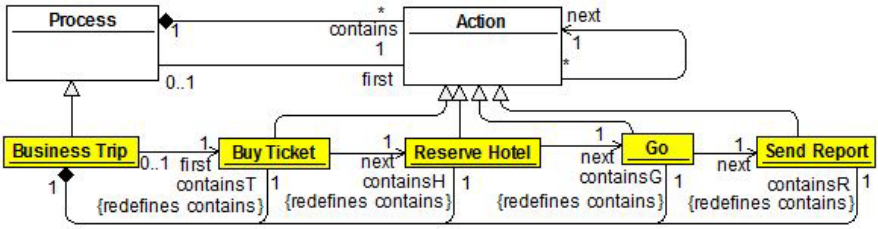

Figure 2. Simple Process metamodel specialization – Business Trip.

The specialization represents a metamodel for Business Trip. Classes of the specialization are no more abstract – they can have instances, e.g., TripToBerlin, ReserveLufthansaFlight etc. The specialization diagram relies on UML *redefines* feature for subclasses – inherited association ends having the same name as for the superclass are

redefined automatically, but renamed ends use the explicit *redefines* modifier. Our intuitive semantics of the specialized metamodel completely complies with the semantics assumed for abstract classes of the generic Process metamodel. We will call the generic metamodel which is being specialized a **Universal Metamodel** (or UMM for short) – see more details on this in the next section. Classes of a UMM will be shown with a white background, but the specialized classes – with a colored one.

In order to make the metamodel examples more readable and compact we use a custom notation for specialized classes and redefined associations here – we show only the specialization and add the original class and role names from UMM in braces (and in bold italic font) – see Figure 3 which presents the same specialization as in Figure 2.

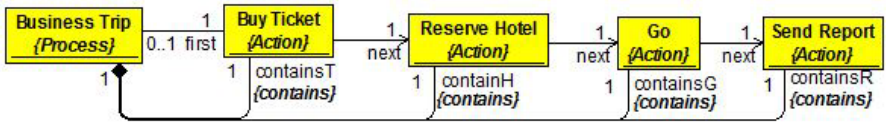

Figure 3. Business Trip specialization in the alternative notation.

Now let us give a more formal definition of metamodel specialization. There may be as many classes specialized from UMM classes as required. But there is a restriction that only a precisely defined set of features enabled by UML *redefines* construct can be applied to inherited from UMM class properties – attributes and association ends. The permitted redefinition includes:

- the definition of a default value of an attribute (but not redefining the attribute type or multiplicity) – syntactically the redefinition is ensured by using the same attribute names in the subclass
- for a redefined association end the multiplicity of the association may be redefined (narrowed), explicit redefinition must be used when a different role name is used for a subclass.

No new (non-redefined) attributes or associations can be introduced in the specialization. Default values of attributes are essential here since they determine that a newly created instance of a specialized class will have just these values. A specialization of a UMM class may also be an abstract one (with the standard UML semantics), if it has a further specialization to non-abstract classes. But concrete classes cannot be specialized further. Specialized classes may have OCL constraints attached.

The goal of these specialization restrictions is to permit only meaningful specializations where subclasses are true specializations of the corresponding UMM classes with a similar, but more restricted meaning.

3. Universal Metamodel and Universal Engine

Now when the permitted metamodel specialization has been defined it is time to try to formalize more deeply the intended meaning of a metamodel by adding some precisely defined behavior to it. We will define this behavior by means of an executable engine named the **Universal Engine** (UE) for the given UMM. By definition of this UE we understand a specification how this UE will work on arbitrary specialization of the UMM. In this sense there is only one unique UE for the given UMM.

We will explain the concept of UE on an example – a UMM in Figure 4 and one of its specializations in Fig.5. We will explain the functionality of UE on just this specialization, but with the goal to understand how the UE will work on any specialization. Since our main domain in this paper is diagram editor definition, Figure 4 represents a UMM for a family of very simple editors which are capable of creating just one diagram consisting of nodes and edges, with the structure specified in a specialization.

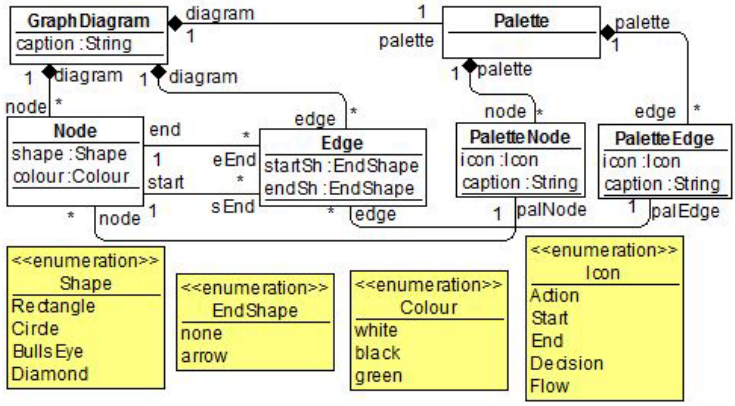

Figure 4. Universal metamodel for simplified diagram editors.

Nodes can have any shape and fill color, edges have color and end shapes. But no text element creation for nodes and edges is present in this simplified version. However one vital element for defining an editor functionality is present here – the Palette together with its elements for creating nodes and edges. Any diagram editor is to be run by a user, but User is not explicitly present in the UMM. Instead, we say that the user can click any palette element – palette node or edge. In response a node or edge instance of the specified kind will be created by UE. Certainly, for a new node the user after the click has to select an empty place in the diagram area, but for a new edge – its start and end nodes. Actually this is all we have to say here on the generic behavior of UE in this simple case, in addition we assume that the editor always starts with an empty diagram with its Palette shown. The details of real behavior of UE for creating a diagram of a specific kind are defined in a specialization of UMM. E.g., only there it is visible what elements will be shown in the palette and what diagram Node specialization will be created by clicking on a PaletteNode specialization. Figure 5 presents one such UMM specialization – a very simple flowchart editor, the custom notation for specialization introduced in Section 2 is used there.

The GraphDiagram class is specialized to a concrete diagram kind – SimpleFlowchart. The specialization contains four specialized node kinds – Action, Start, End and Decision as subclasses of the UMM Node class and just one Edge specialization – Flow. Accordingly, the Palette is specialized to FlowchartPalette containing four PaletteNode subclasses (one for each node kind) and one PaletteEdge subclass. Default values are assigned to all inherited attributes in subclasses, for node and edge subclasses these are the default style attributes to be used by UE when a node or edge instance is created (note the different shapes for all node kinds). For palette elements different icons and texts are specified accordingly. All UMM associations are redefined as well – using the automatic redefinition where association end names are the same for the superclass and subclass and explicit redefinition otherwise.

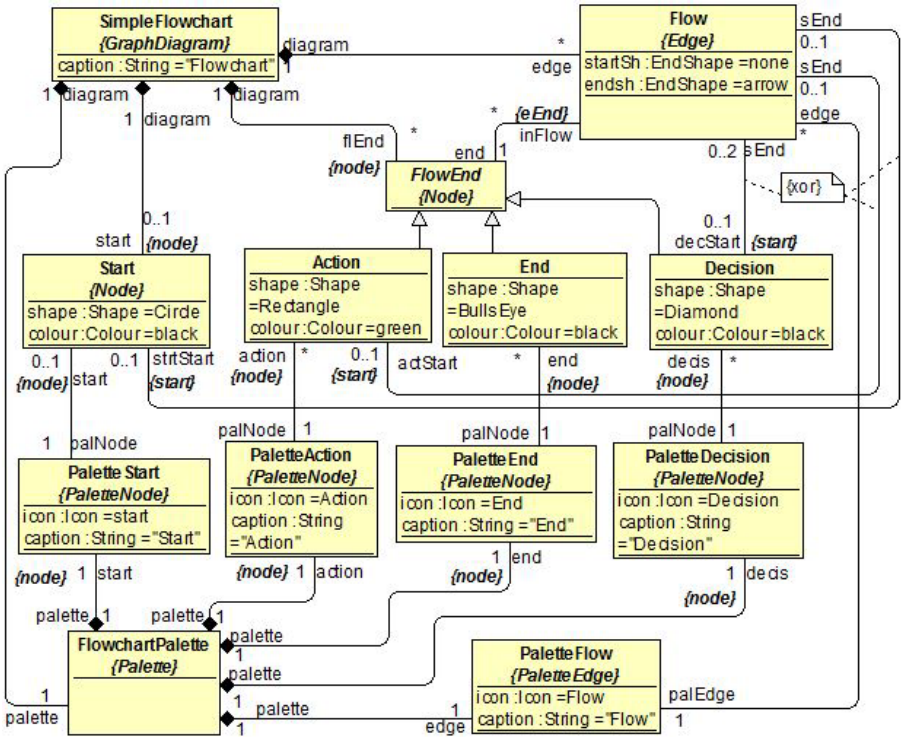

Figure 5. UMM specialization defining a simple flowchart editor.

To reduce the number of associations in the specialization an abstract subclass is used – the FlowEnd class. It groups together the concrete subclasses Action, End and Decision, which have a common containment association in the SimpleFlowchart diagram and a common association to incoming Flow instances. The fact of specializing the Node class in UMM is also shown in the FlowEnd class, the subclasses inherit it. Note that the specialization within the specialized metamodel is shown via the traditional UML notation. The fourth Node subclass – the Start couldn't be included in the group since it has no inFlow association to the Flow, in addition the containment multiplicity is different. No grouping is used for the outgoing flow specification since the multiplicities are too different, instead the simplest UML {xor} constraint is used to specify that a Flow can start from only one Node subclass instance. Note that each PaletteNode subclass has a specialized association to the corresponding Node subclass – PaletteAction to Action and so on, thus enabling the appropriate node type creation upon a palette element click.

This relatively simple specialization contains no explicit OCL constraints. However multiplicities and {xor} constraints act as required according to the UML standard. Thus only one Flow can exit an Action or Start node, and no more than two flows a Decision node, only one Start node can be created per flowchart and so on. If the user tries to violate these constraints UE shows a fixed error message – "Action not permitted". Thus a relatively rich diagram editor definition can be obtained by a simple generic UE specification and an appropriate UMM specialization. Though the behavior of UE was explained just on the specialization for the simple flowchart, it should be clear how it would behave on any correct specialization of UMM in Figure 4. We note here

that many purely technical functions (e.g., diagram layout management) to be performed by UE do not depend on the given specialization and can be performed by components of UE based solely on the UMM – see more in Section 7.

4. Application of Metamodel Specialization to Diagram Editor Definition

The previous section gave a simple introduction into basic concepts of UMM and UE for graphical diagram editor definition. Our goal in this paper is to define a platform for realistic diagram editor definition by means of metamodel specialization. The capabilities of such editors should be similar to our previous editor definition platform TDA [7,8]. The UMM will provide a general schema for any such editor – be it an editor for flowcharts, for UML class diagrams, for UML activity diagrams etc. The generic behavior of all such editors will be defined by the Universal Engine (UE) operating on UMM. But making the editor behaving just as a Flowchart editor should be made by defining an appropriate specialization of the UMM – then the UE will act as a true Flowchart editor. The UMM will provide a vision of such a diagram editor – on what concepts it is operating (see the UMM in Figure 6). We will consider only diagram editors for pure graphical modelling purposes – without the need to generate some code from the diagram, to run an interpreter on it etc.

The UMM for our diagram editor domain provides a generic data schema on which the behavior of UE and thus any specialized editor is based. But the behavior dynamics involves also the editor user whose actions actually determine the result. The previous section introduced one element for interaction with the user – the diagram Palette, but there are more.

The possible user actions will not be explicitly captured as UMM classes but they will be tied up to most of UMM classes. The semantics of UE behavior will be defined just in terms of these actions – what happens if the user clicks a palette element, double-clicks an existing diagram node, enters a compartment value as a text input etc. But there is a strict requirement that all results of a user interaction must be stored as instances of appropriate UMM classes – more precisely, of their specializations.

Now let us explain our vision of such diagram editors and their potential behavior on the basis of the UMM in Figure 6. Typically any real diagram editor, including those defined via TDA, contains the concept of Project – a set of related diagrams having a common usage. Therefore we also include Project class in our UMM. 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. The concepts of Graph diagram, Node, Edge were introduced already in Section 3, only some more attributes are added to these UMM classes. Certainly, we will not present completely all used in practice diagram style attributes, but only the main ones. The main new concept in this UMM is the Compartment – an element in a node or at an edge containing some text. The value attribute of a compartment shows the string really displayed in a diagram. But there are a lot of other attributes specifying the structure of texts, the means for entering them by a user, a way and style of displaying them and so on, in addition these attributes differ for texts in nodes and at lines, therefore we have NodeCompartment and EdgeCompartment classes. A compartment text may have a substructure – e.g., a class attribute text in UML consists of its name, type, default val-

ue, modifiers etc., these elements are separated by constant prefixes or suffixes in the common string value, the order of concatenation may be defined by the subCompNo attribute if required. But during the value creation by the user they typically are processed as separate compartments. This structuring in the UMM is supported by the parentCompart – subCompart association, permitting each part to be processed separately as a subcompartment.

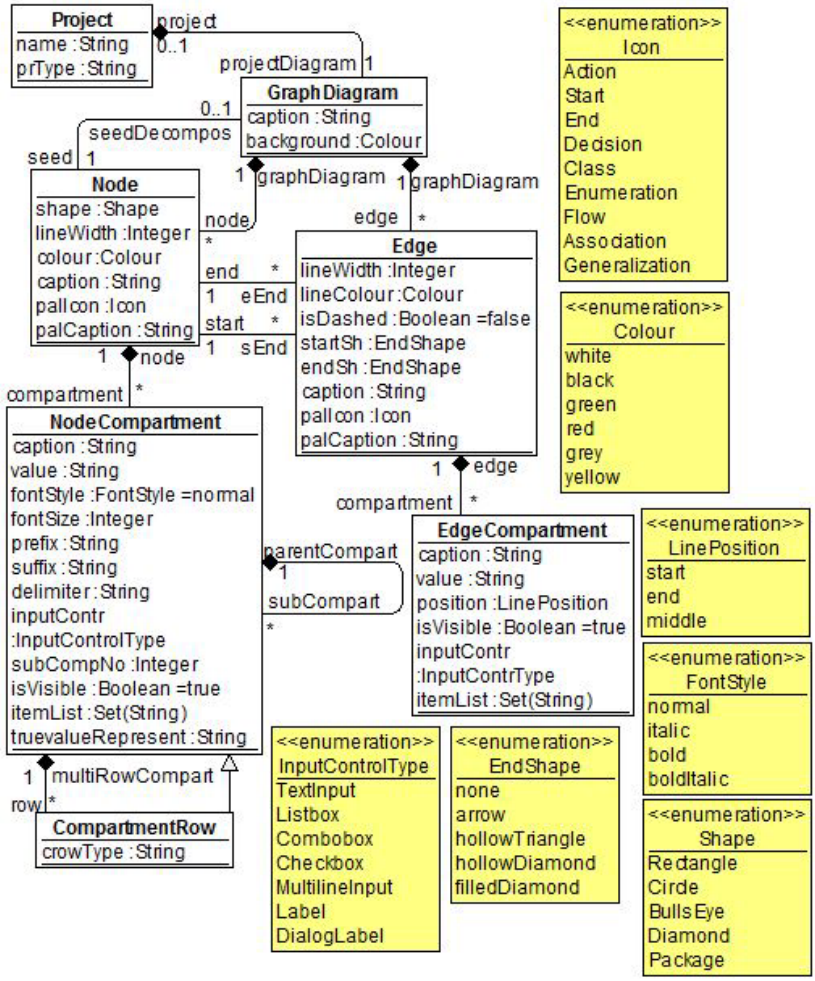

Figure 6. UMM for real diagram editors.

Further, the inputContr attribute determines the input control type, which is offered to the user for entering the compartment or subcompartment value. Certainly, the supported types of input controls depend on capabilities of UE, but the minimum list includes simple text input, checkbox for entering Boolean values and listbox or combobox for offering to the user a list of values to select from (in case of combobox a direct value input is also permitted). For both these controls there must be a possibility to define the appropriate value list, therefore the itemList attribute of type Set(String) is added. The default value of this attribute must be set in the specialized compartment

class (if listbox or combobox is selected for the compartment input), this value may be a constant set or an OCL expression deriving the set from other diagram elements already created. One more nontrivial input control is multiline input containing rows with the same properties (such as the whole attribute list for a class), there a special CompartmentRow subclass permits to process each line separately and add a new line (a line may be a structured compartment as well).

The whole input process of a compartment is organized by UE in a fixed way, the specialization may configure only a compartment structure, the input control used for a compartment/subcompartment entry and provide the required values. In addition, a standard UML constraint (an OCL expression based on elements of the specialization and returning a Boolean value) may be added to the specialized compartment class to check the entered compartment value after the user has completed the input of this compartment. Some examples for these rather complicated specialization features are demonstrated in Section 6 on class diagram editor fragments.

A significant difference of this UMM from the UMM in Section 3 is that there are no more palette-related classes in UMM. Even the simplest specialization for an editor in Figure 5 had to contain two classes for each node (or edge) kind – for node itself and the related palette element, which can make the specializations quite large and cluttered for real diagrams. The only essential information for palette elements is the icon and caption attributes. Now these attributes (palIcon and palCaption) are added to Node and Edge classes. These attributes must have a default value for each non-abstract Node or Edge subclass in a specialization. Then the slightly extended UE can automatically generate a correct palette for each diagram kind in the specialization.

We conclude this section by a rather informal description of UE related to this UMM. Upon start the UE permits the user to create a new diagram editor project of the kind defined by the current specialization (a flowchart project, a class diagram project etc.) or open an existing project of this kind. After that the project diagram (either empty or already filled) with its (generated) palette is shown. The user can add a new diagram of a supported kind via creating its seed from the palette, or open an existing diagram – by double-clicking on the seed. The diagram is opened together with its palette, and then new diagram elements can be added in the manner described in the previous section. But a new possibility is to enter compartment values which are defined in the specialization – the compartment editor (a dialog form) is opened after a new node or edge is created. The compartment editor can be opened also for existing nodes or edges by a double-click. Besides this specialization-related UE behavior, UE offers some default behavior to the user – to save a project, to modify the default style of a node or edge, to modify the layout etc.

5. The Real Flowchart Specialization Example

In this section the new possibilities of UMM and UE are demonstrated on a more realistic flowchart editor example – see Figure 7. The Project class from UMM is specialized to FlowchartProject with just one FlowchartProjectDiagram attached to it. This diagram contains named FlowchSeed nodes from which the corresponding Flowchart diagram instance can be opened. There are no more palette-related classes, instead all Node and Edge specialization classes contain the palette-related attributes. This means that UE can generate all relevant palette elements automatically. Thus the palette for FlowchartProjectDiagram contains just the element for FlowchSeed node creation.

When the user clicks this element, a new seed node is created. In order to have a user-defined name for a seed (and the related flowchart as well), the FlowchNameCompart class (specialized from the NodeCompartment) is associated to FlowchSeed . Only the caption and inputContr attributes with their default values appear in the compartment specialization – other attribute values are not required for this simple case.

The generated palette for the Flowchart diagram itself contains four node elements (same as the explicit palette in Figure 5) and two edge elements (for both kinds of Flows). The Flowchart node definitions in this specialization are also similar to those in Section 3, only a name can be created for an Action and a condition text for a Decision – both use simple TextInput fields for value input.

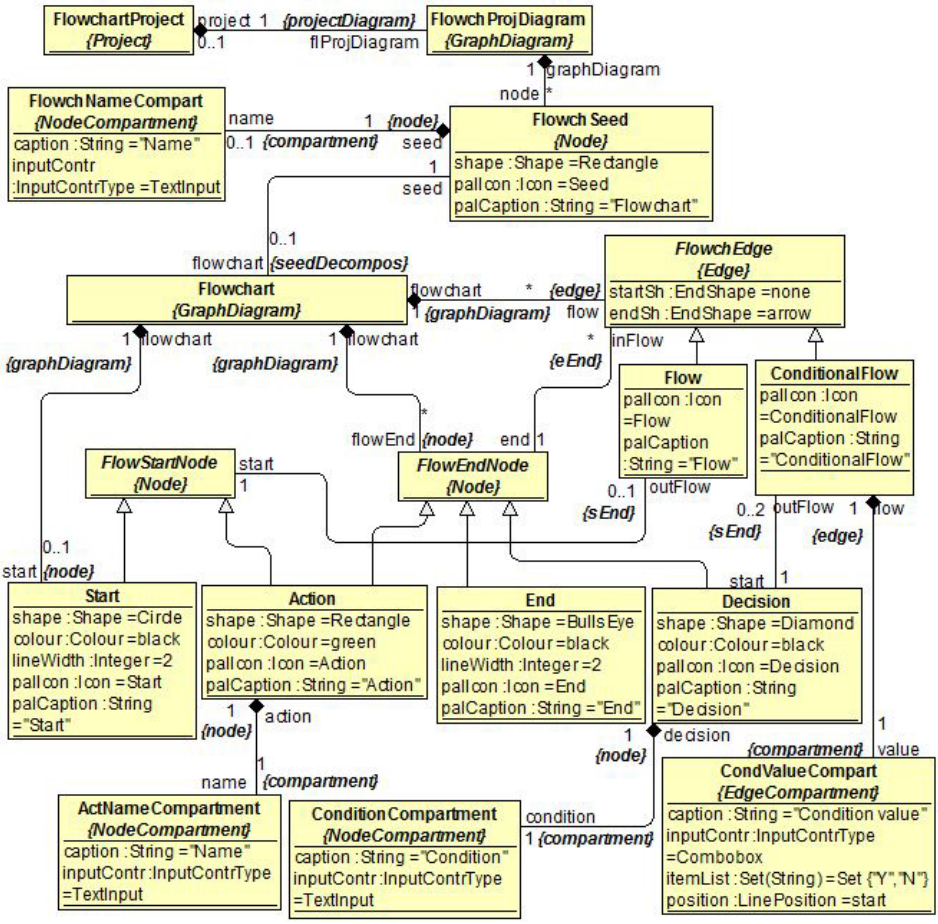

Figure 7. Realistic flowchart editor specialization.

But here two edge subclasses are defined – the Flow as in Section 3 and the ConditionalFlow which can exit only a Decision node (no more than two instances). A conditional flow has a text attached near to its start – typically Y or N but any other text can be used as well. Therefore in the specialization the class CondValueCompart (a subclass of EdgeCompartment) is used. A different input control type – Combobox is used there, and for this control also the default value for itemList attribute must be defined – here

it is a constant set of strings – Set {"Y","N"}. The position attribute specifies that the entered text has to be positioned near the start of the edge.

Note also the abstract class FlowchEdge in the specialization, with the concrete subclasses Flow and ConditionalFlow, which holds the common attributes and associations for both Flow and ConditionalFlow..

Thus this example shows that using only basic UML class diagram constructs such as multiplicity a UMM specialization can define relatively complicated editor behavior. In order to obtain a more compact specialization, it is assumed that some more compartment style attributes (in addition to those shown in Figure 6) have their default values set already in UMM, e.g., fontStyle is set to the value *normal*.

6. Fragments of Class Diagram Example

In this section some basic fragments containing new features for a specialization of the same UMM in Figure 6 defining a class diagram editor will be given. The functionality of the editor is approximately that used for creating class diagrams (metamodels) used in this paper. The whole specialization can be defined in 3 class diagrams each to be shown in an A4 page in a readable way. The supported node types are Class and Enumeration, but edge types – Association and Generalization. In addition, the UML package mechanism is included, but in a slightly non-standard way – using Package diagrams. The main new elements in this specialization are the use of compartment rows, subcompartments and OCL expressions for default values and constraints. Figure 8 presents a fragment of this specialization showing the Class node and some of its compartments: Class name compartment, IsAbstract compartment and Attribute compartment. The IsAbstractCompartment enables the user to enter the Boolean value specifying whether the given class is abstract or not; this is done via a checkbox control (the value is stored in the model as a string "true" or "false"). This value is not visible directly in the class node – therefore the attribute isVisible is set to false (according to the UML standard). Instead, the value is displayed by setting the appropriate style for the class name compartment (italic if the class is abstract). To specify this setting, the value of fontStyle attribute of ClassNameCompartment is set by an OCL expression:

```
fontStyle : FontStyle = if self.class.isAbstract
.value.toBoolean() then italic else bold endif
```

(the expression is not shown in Figure 8 to reduce the box size). The Attribute compartment is to be created by the user via a specific control – MultiLineInput. This control is specially adjusted to creating texts consisting of logically independent lines, such as attributes or operations in a Class node. Therefore the UE provides an independent entry of each line using the CompartmentRow class in UMM which is specialized here to AttributeRow. Further, the line can have a complicated substructure: each attribute has a name, type, default value, multiplicity etc. This is supported in UE by the subcompartment concept (see Section 4). Here we have the AttributeName, AttributeType, DefaultValue etc. subcompartments (only the first two are shown in Figure 8). To specify how the values are to be concatenated to a common string, prefixes or suffixes are used – see the prefix ":" for the type. Each subcompartment can be entered using a specific control – the type is entered using a combobox offering the most typical values (primitive types and all defined enumerations in the model). Therefore the itemList attribute of this compartment is specified by the OCL expression:

```
itemList : String[*] = set{"Integer","Boolean", "String",
"Real"} -> union(Enumeration.allInstances().name.value ->
asSet() )
```

The allInstances OCL construct iterates over the whole project, thus all Enumeration instances are collected and their name values included in the list. A similar but more complicated expression can be defined to offer all already existing class names when a class name is to be entered (in order to create occurrences of a class in several diagrams). Similar expressions preparing a typical value list can be defined for other subcompartments as well, e.g., for type-dependent default value prompting.

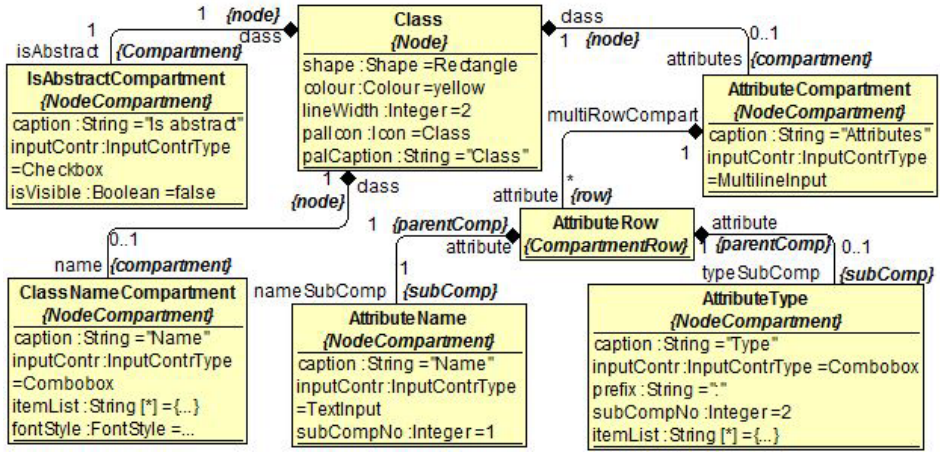

Figure 8. Fragment of class diagram specialization – class attributes.

Finally, we show an example of OCL class constraint to be used for checking the entered compartment value correctness – the class AttributeName has a constraint specifying that attribute names must be unique for a class:

```
{self.attribute.multiRowCompart.attribute.nameSubComp.
value -> forAll(val | val<>self.value)}
```

Such constraints have to be specified directly for the corresponding compartment class, then UE knows that the constraint has to be evaluated right after the user has exited the corresponding input control.

To conclude, the use of OCL permits to obtain at least the same functionality of a class diagram editor as can be defined using custom transformations at extension points in TDA and close to many commercial UML editors.

7. Implementation Principles

The planned implementation of metamodel specialization based DSML editor platform really consists of two parts – the Base UE (including DSML project management, Presentation engine for diagram drawing and layout management, Dialog engine for current form building and user input processing) and the UMM Specialization engine. All components of the Base UE can in fact work in terms of the UMM. However, real instances in a running diagram editor are solely for classes and associations in the giv-

en specialization. These instances are created and managed in a repository by the Specialization engine (SE), which is the sole component using and navigating the specialized metamodel. For example, when a new Node subclass instance is being created, the SE sets the default values of the redefined attributes and collects all specialized compartments for this Node subclass. Now this information has to be passed to the dialog engine (DE) for displaying the input form. However, as stated before, the DE should work in terms of UMM. To achieve this, a temporary instance repository (of limited size) according to the original UMM can be used. Since all specialization classes are true subclasses of UMM classes, at instance level such interpretation can be easily performed by SE. Similarly, at instance level all links corresponding to redefined associations can be interpreted as links for associations in UMM, at instance level no name conflicts appear. Certainly, here UMM has to be extended by a reference attribute to the specialized element, but this attribute is used only by SE. Thus DE can get the information for creating the input form according to UMM and store the obtained user input also in this UMM-based repository. The next steps are performed by SE, which checks the data and then invokes the Presentation engine (PE) for drawing the node in the current diagram. PE again does this in the context of UMM, since specific attribute values (e.g., for style) have been set by SE. Here it is clear that no more than instances in the current diagram must be maintained by SE in this temporary UMM-based repository. In addition, there must be a component for interpreting the OCL expressions (for default values and constraints, referencing the specialized model). If Eclipse EMF is used as a model repository there is a freely available OCL interpreter [18]. For other repositories the solution used in TDA can be applied – use the Lua/lQuery [11] instead of OCL, this language has sufficient expressive power and is implemented within TDA for several repositories. The components of Base UE (DE, PE) in fact are quite similar to those existing in the TDA implementation, therefore they could be reused. Thus the effort for implementing the proposed approach could be much lower than that used for TDA, the only new component would be SE.

8. Conclusions

The analyzed DSML examples show that the proposed approach for DSML editor definition based on metamodel specialization has a number of advantages and is usable in practice. The specialized metamodels which use only basic UML class diagram features and OCL for more complicated situations are natural formalizations of the graphical syntax of the DSML to be defined. Thus such metamodels are sufficiently easy to create and read. To a great degree this fact shows up when compared to the corresponding DSML definitions in the existing TDA platform. For very simple DSMLs such as the simple flowchart where the type instances in TDA can be created using the Configurator without any extension points the efforts are comparable. But for Class diagram editor a significant use of extension points and transformations is required in TDA, with a large effort required to create these transformations due to the complicated runtime metamodel in TDA. Thus the complete DSML definition there is in fact invisible, the type metamodel instances provide only a graphical skeleton of class diagram definition. At the same time the definition using metamodel specialization describes the supported functionality in a very explicit way. Creation of required OCL constraints is also quite straightforward since the specialization directly defines also the runtime metamodel to be referenced in constraints. If more features are to be added, the specializa-

tion extension by new specialized classes is also very straightforward. Most probably, if TDA would be created now, the metamodel specialization approach would be used. In addition, the metamodel specialization can be used for direct diagram syntax definition (in a simpler way than the current DD standard [15]), see more in [17].

Acknowledgements

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

References

[1] Eclipse. http://www.eclipse.org. Accessed August 2016.

[2] Graphical Modeling Framework (GMF, Eclipse Modeling subproject), http://www.eclipse. org/gmf/. Accessed August 2016.

[3] Obeo Designer: Domain Specific Modeling for Software Architects, http://www.obeodesigner.com/. Accessed August 2016.

[4] EuGENia Live, http://eugenialive.herokuapp.com/. Accessed August 2016.

[5] S. Cook, G. Jones, S. Kent and A.C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*, Addison-Wesley, Boston, 2007.

[6] S. Kelly, J.P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley, Hoboken, 2008.

[7] J. Barzdins, E. Rencis and S. Kozlovics, The Transformation-Driven Architecture In *Proceedings of DSM'08 Workshop of OOPSLA 2008*, Nashville, Tennessee, USA, pages 60-63. 2008.

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

[9] J. Barzdins et al., GrTP: Transformation Based Graphical Tool Building Platform, *Proceedings of MDDAUI'07 Workshop of MODELS 2007*, Nashville, Tennessee, USA, CEUR Workshop Proceedings, http://ceur-ws.org, **297** (2007), 4 pp.

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

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

[12] A. Sprogis and J. Barzdins, Specification, Configuration and Implementation of DSL Tool. In *Frontiers of AI and Applications* **249**, Databases and Information Systems VII, IOS Press, pages 330-343, 2013.

[13] E. Rencis, J. Barzdins and S. Kozlovics, Towards open graphical tool-building framework. In *Proceedings of BIR 2011*, RTU Press, Riga, pages80-87, 2011.

[14] S. Pierre, et al., A Family-Based Framework for i-DSML Adaptation. In *Proceedings of 10th European Conference ECMFA 2014*, *LNCS* **8569**, Springer, pages 164-179, 2014.

[15] *Diagram Definition (DD)*, Object Management Group, version 1.1 – formal/2015-06-01, 2015.

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

[17] A. Kalnins and J. Barzdins, Metamodel Specialization for Graphical Modeling Language Support, accepted for *MODELS 2016*.

[18] Eclipse OCL, https://projects.eclipse.org/projects/modeling.mdt.ocl. Accessed August 2016.