Software & Systems Modeling (2019) 18:1699–1735
https://doi.org/10.1007/s10270-018-0668-3

SPECIAL SECTION PAPER

Metamodel specialization for graphical language support

Audris Kalnins[1] · Janis Barzdins[1]

Received: 14 March 2017 / Revised: 19 January 2018 / Accepted: 13 February 2018 / Published online: 2 March 2018

Abstract
Most of current modeling languages are based on graphical diagrams. The concrete graphical syntax of these languages typically is defined informally—by text and diagram examples. Only recently, starting from UML 2.5, a formalism is offered for defining the graphical syntax of UML. This formalism is based on Diagram Definition standard by OMG, where the main emphasis is on enabling diagram interchange between different tools implementing the given language. While this is crucial for standardized languages such as UML, this aspect is not so important for domain-specific languages. In this paper, an approach is offered for a simple direct definition of concrete graphical syntax by means of metamodels. Metamodels are typically used for a language definition, but mainly the MOF-inspired approach via meta-metamodel instantiation is used. We offer an alternative approach based on core metamodel specialization which leads to a more direct and understandable definition staying at the same meta-layer. In addition, our approach permits a natural extension—facility for a graphical editor definition for the given language, which is vital in the world of DSLs. In contrast to most DSL development platforms, which are based on the abstract syntax metamodel of the language and a mapping to graphics, our facility is based directly on the graphical syntax. But we show that in those cases where the relation to the DSL abstract syntax is really required, a mapping from the graphical syntax to abstract syntax can be relatively easily defined by the specialization approach.

Keywords Metamodeling · Metamodel specialization · Graphical syntax definition · Graphical DSL · Graphical editors

1 Introduction

Metamodels are the most used formalism for defining graphical modeling languages. The four-layer metamodeling approach defined by OMG MOF [1] is used for the definition of nearly all graphical modeling languages maintained by OMG. Certainly, the most notable such language is UML [2], but there are many other languages. The layer M3—MOF itself, is used to define M2 artifacts—metamodels for concrete languages, e.g., UML. This is done by instantiation—metamodels in M2 are defined as instances of MOF as a meta-metamodel. Metamodels in M2 are meant to define an abstract syntax of a language. The UML metamodel specifies the abstract syntax of UML—the modeling concepts, their attributes and relationships, as well as rules for combining them. The abstract syntax of any valid UML model must be an instance of this metamodel. However, the concrete graphical syntax of UML diagrams is defined completely informally, frequently on the basis of examples.

Starting from UML 2.5 [2], some formalization is offered also for defining the graphical syntax of diagrams. It is based on the new OMG standard for Diagram Definition (DD) [3]. However, the main goal of this formalization is to enable a diagram interchange (DI) between modeling tools, rather than a simple and precise diagram syntax specification for tool developers. Therefore, the first component of the DD approach is the DI metamodel which permits to define the structure of a diagram in a very abstract way, with the goal just to specify which diagram elements should be interchanged between tools. The second component of DD is the DG (Diagram Graphics) metamodel which is oriented toward a low level specification of graphical element rendering in a tool. The elements of DI must be mapped to elements of DG. The OMG DD approach is discussed in greater detail in Sect. 4.

Communicated by Dr Jörg Kienzle and Alexander Pretschner.

✉ Audris Kalnins
audris.kalnins@lumii.lv

Janis Barzdins
Janis.barzdins@lumii.lv

1 Institute of Mathematics and Computer Science, University of Latvia, Raina bulvaris 29, Riga, Latvia

Thus, the problem of simple but at the same time precise definition of the graphical syntax for a graphical modeling language is still open. This problem is very significant now when, besides standardized modeling languages, so many domain-specific modeling languages are defined and used.

In fact, two paradigms for concrete graphical syntax definition by means of metamodels are available. One—the most used so far—is the metamodel instantiation which was already mentioned in the context of MOF. This paper proposes another possibility—the specialization of a base metamodel to describe the syntax of a concrete language. Here the resulting metamodel just contains subclasses of the base metamodel. This approach does not require another meta-layer for the resulting metamodel. The specialization is based on standard UML features—creating subclasses of the base metamodel, redefining (or subsetting) class properties and adding new OCL constraints. Section 6 describes the basic principles of the proposed specialization approach. The main goal of the paper is to show the usage of metamodel specialization for the graphical language syntax definition and related tasks (language editor definition) and to analyze the advantages of this approach when compared to the traditional instantiation. In particular, these advantages are based on an appropriate choice of the base metamodel to be specialized—the Universal Metamodel (UMM) for the given task.

A task closely related to the diagram syntax definition is the creation of platforms for building graphical editors for such languages. The approach used in most of these platforms is a sort of mapping the abstract syntax (or domain model) of the language (defined via MOF facilities) to a graphical notation. First and foremost, these platforms are based on Eclipse GMF [4], where the abstract syntax is defined via an EMF [5] metamodel, the graphical concrete syntax is defined via GMF graphics metamodel, and the relation between the two metamodels is defined via the mapping metamodel. A detailed analysis of the Eclipse GMF approach is provided in Sect. 2. Many other platforms use a similar approach based on the domain model of the language—see more in Sect. 5.

Only a few approaches for defining diagram editors directly on the basis of their graphical syntax exist. The platform devoted most directly to graphical DSL tool (graphical modeling language editor) definition is the platform developed by IMCS UL—TDA (Transformation-Driven Architecture) [6,7]. There a fixed Tool definition metamodel is proposed, where the concrete syntax of a DSL and an editor for it is defined as an instance of this metamodel (thus, the traditional instantiation approach is used there). A detailed analysis of the TDA approach is given in Sect. 3.

This paper shows that not only a rich class of graphical languages, but also quite advanced graphical tool building platform can be defined in a relatively easy way via metamodel specialization. The proposed metamodel for editor building is a direct extension of the metamodel used for language syntax definition. Sections 7–12 provide the details of the metamodels to be specialized and specialization examples for languages and editors.

This paper is an extended version of the conference paper [8]. First, a deeper analysis of the existing traditional instantiation-based approaches to language and tool definition (Eclipse GMF, TDA, OMG DD, etc.) is added in Sects. 2, 3, 4, and 5. On the basis of this analysis, the new Sect. 13 now provides a significantly deeper comparison of our approach to the traditional ones. In addition, we provide a new section describing how for those cases where the abstract syntax of the DSL is required as well the editor definition can be extended by a declarative definition of the mapping from the graphical to abstract syntax using the same specialization approach. The section on implementation principles is significantly extended as well, by providing internal metamodels and other implementation details.

Sections 2, 3, 4, and 5 provide some details and also problems of the traditional approaches. Section2 gives a brief overview of the graphical language definition via Eclipse GMF framework [4]. The overview is based on a very simple graphical language —flowchart example (the example is reused in subsequent sections as well). Models required for the definition of the language editor are shown. On this basis, it is shown that, though the GMF framework permits to build support for various languages, even for this very simple language the definition is not so trivial. Section 3 briefly revisits a TDA platform approach for graphical syntax definition using metamodel instantiation. Section 4 provides a brief analysis of the OMG DD approach to language definition. Section 5 briefly comments on other instantiation-based approaches.

Sections 6–12 provide the details of the metamodel specialization approach and examples of its usage for graphical language and tool definition. Section 6 presents the basic principles of the proposed metamodel specialization. Section 7 describes the Universal Metamodel for graphical syntax definition and gives an example of its specialization for the complete precise flowchart syntax definition. Section 8 shows how the Universal Metamodel can be extended and connected with a Universal Engine to provide the basis for graphical editor definition. The complete Flowchart editor definition and an essential fragment of a simplified class diagram editor is presented in Sects. 9 and 10, respectively. Section 11 explains how the Universal Metamodel and the Universal Engine can be extended to provide also a synchronous building of the abstract syntax notation of the DSL model. Section 12 presents basic ideas of the implementation. Section 13 provides a deeper comparison of the specialization approach to traditional ones and shows its advantages. The conclusion presents some future use cases of our approach.

2 Brief overview of graphical language definition in Eclipse GMF framework

In this section, we provide a brief overview of the very popular Eclipse GMF framework [4]. This framework for a long time has been the most popular environment for building tools for graphical DSLs. It is a very classic example of the metamodel instantiation approach for tool building. GMF is based on the classical paradigm for graphical language definition, where the definition starts from the abstract syntax or domain model for the given language. This model is built as an instantiation of the Eclipse EMF [5] metamodel. All semantic aspects of the language should be built into this domain model, using OCL constraints for domain elements. Other three models are required to build the graphical and tooling aspects of the language. The graphical syntax of the language is defined via graphics model which is an instance of the GMF Graphical Definition (GMFGraph) metamodel. The model describes the graphical syntax of the language in terms of nodes, connections, compartments and labels. However, for these diagram elements many graphical aspects are detailed in terms of figures, which in fact are taken from the lower-level Eclipse framework GEF [9], on which GMF is based. The GEF framework is not so model based, it is more oriented toward direct Java APIs. Therefore, defining new kinds of figures, as a rule, requires programming in Java. The editor functionality aspects are defined by the tooling model which is an instance of the GMF Tooling metamodel. However, only palette and menus can be defined there. The details of diagram element property editing should be defined already at the domain model (EMF) level, by customizing the generated Edit and Editor components, mainly at Java level; therefore, we do not show this editing aspect in greater detail. The graphics and tooling models are linked to the domain model via the mapping model which is an instance of the GMF Mapping metamodel. The mapping model defines in a hierarchic way the graphical elements by which each domain element should be visualized. Some more complicated mapping situations (e.g., involving a choice depending on element properties) can be detailed using OCL constraints. In addition, for the main elements the related tooling elements from the tooling model are also shown in the mapping model. The GMF framework provides only one way to obtain a working graphical editor for the language—generate Java code (resulting in an Eclipse plug-in). The code generation can be customized by using one more model—generation model which is an instance of the GMF Generation metamodel. The generation model is derived from the mapping model, and it references all four models described before. Some customizing can be done at this model level, but more complicated customizing typically involves programming in Java.

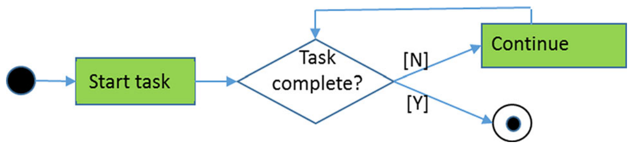


Fig. 1 Flowchart example

The domain, graphics and mapping models together to some degree can be considered as a graphical syntax definition of the language, but in practice this aspect is rarely used because of being complicated and not very readable. It is one of the reasons why an independent diagram graphical syntax definition facility (DD) was proposed by OMG.

To explain the basic ideas, in this paper we introduce a simple graphical language example—flowchart, on which all language and tool definition approaches will be demonstrated and compared. Now a more detailed explanation follows. Figure 1 shows a simple example of a flowchart.

The flowchart contains 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 (an informal text). Besides, 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, and no more than two conditional flows can start from a decision node (they must have different condition value labels). Any number of flows or conditional flows can enter a node (except the start node). And there may be only one start node per flowchart. All actions in a flowchart must have unique names. There may be one more kind of a node not shown in the example—RefinedAction, referencing another flowchart instance. This kind of action is shown in another color (red), but its main functionality appears in the editor as a navigation facility to the referenced diagram.

Now the definition of this simple flowchart language will be shown in GMF. All five models mentioned in the brief GMF overview above, in fact, are true class diagrams (at MOF M1 level, being instances of the corresponding metamodels at M2). However, in the GMF framework user interface all models are natively created in the model tree form. Only the domain model can be easily visualized as a class diagram. Therefore, all five models for the flowchart definition will be shown here in this GMF tree format. Figure 2 shows the domain model. For such a simple language, its domain model (abstract syntax) is in fact one-to-one to its graphical syntax.

The Flowchart class represents a flowchart itself, which contains all other elements. Unfortunately, in the tree view it is not visible which associations (shown as references to the corresponding class) are containments (it is visi-

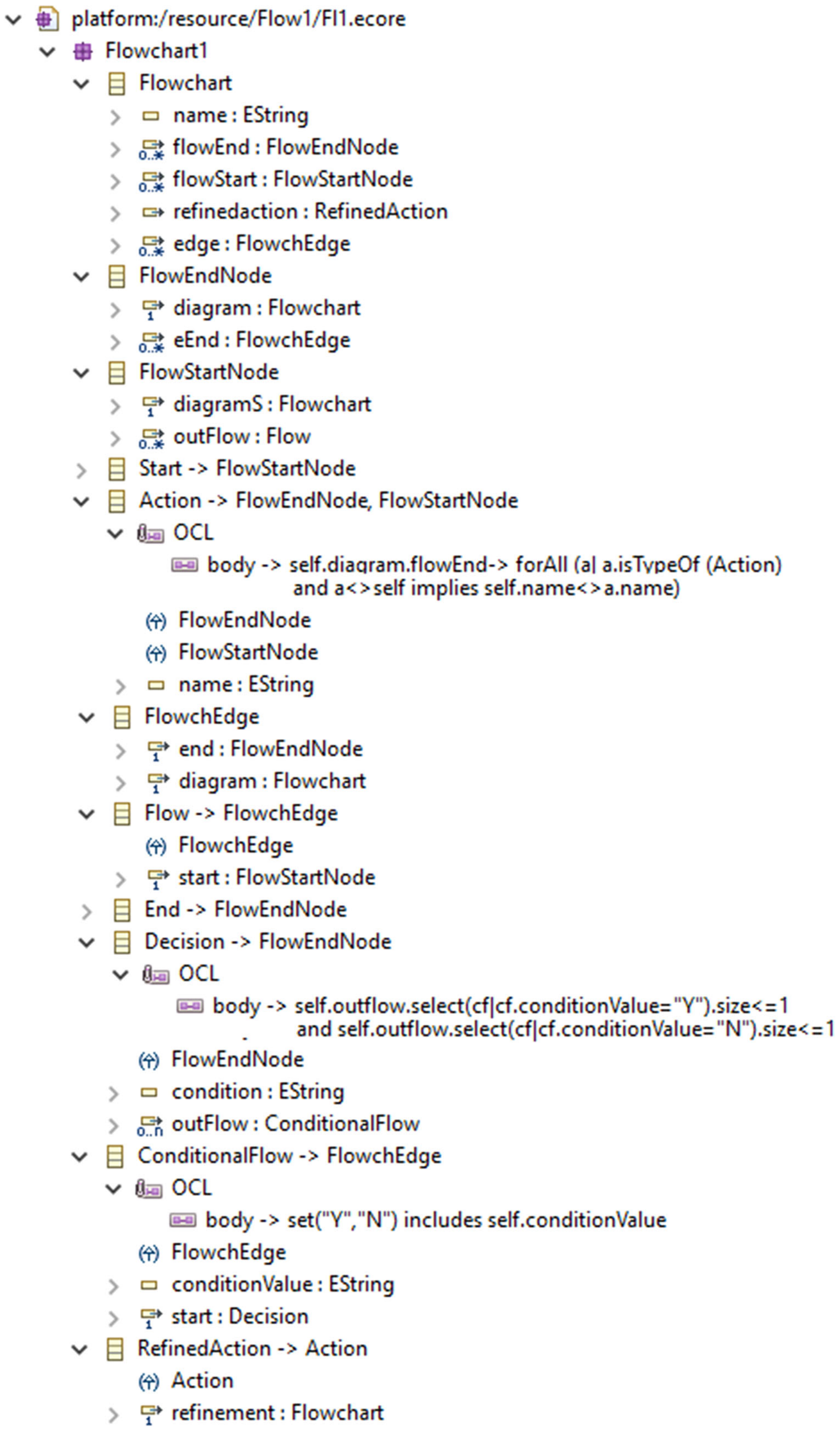


Fig. 2 Domain model of the flowchart (in the tree form)

```
platform:/resource/Flow1/Fl1.ecore
  Flowchart1
    Flowchart
      name : EString
      flowEnd : FlowEndNode
      flowStart : FlowStartNode
      refinedaction : RefinedAction
      edge : FlowchEdge
    FlowEndNode
      diagram : Flowchart
      eEnd : FlowchEdge
    FlowStartNode
      diagramS : Flowchart
      outFlow : Flow
    Start -> FlowStartNode
    Action -> FlowEndNode, FlowStartNode
      OCL
        body -> self.diagram.flowEnd-> forAll (a| a.isTypeOf (Action)
                and a<>self implies self.name<>a.name)
      FlowEndNode
      FlowStartNode
      name : EString
    FlowchEdge
      end : FlowEndNode
      diagram : Flowchart
    Flow -> FlowchEdge
      FlowchEdge
      start : FlowStartNode
    End -> FlowEndNode
    Decision -> FlowEndNode
      OCL
        body -> self.outflow.select(cf|cf.conditionValue="Y").size<=1
                and self.outflow.select(cf|cf.conditionValue="N").size<=1
      FlowEndNode
      condition : EString
      outFlow : ConditionalFlow
    ConditionalFlow -> FlowchEdge
      OCL
        body -> set("Y","N") includes self.conditionValue
      FlowchEdge
      conditionValue : EString
      start : Decision
    RefinedAction -> Action
      Action
      refinement : Flowchart
```

ble only in the properties dialog of the tree element). To make the definition of permitted relations between nodes and edges (connections in GMF) easier, three abstract superclasses are introduced in the model—node superclasses FlowStartNode and FlowEndNode and an edge superclass FlowchEdge. Using this principle, no OCL constraints are required to define the node/edge multiplicity constraints in the Flowchart. There are three explicit OCL constraints present in this domain model—they represent the true semantic constraints present in the informal flowchart definition. Thus, all required semantic constraints of a flowchart can be defined in the domain model.

Figure 3 shows the graphics model for the flowchart.

The initial part of this model shows all figures used in the definition. Figures are contained in a Figure Gallery. The default gallery in GMF is not very rich—there is a rectangle, a rounded rectangle, an ellipse and a polygon for nodes and polyline for edges (connections). Therefore, e.g., the "bull's eye" figure typically used in flowcharts and in activity diagrams for process end must be modeled as an ellipse inside an ellipse, but the diamond figure for decisions as a generic polygon with corner points explicitly specified. For figures, their preferred size and fill color can be specified (other properties which are not used here can be seen only in the properties dialog). Texts inside nodes and at edges are defined as label figures (simple rectangles containing the text). The final part of the tree contains the list of nodes, connections and labels in the diagram, and nodes have references to the corresponding figures.

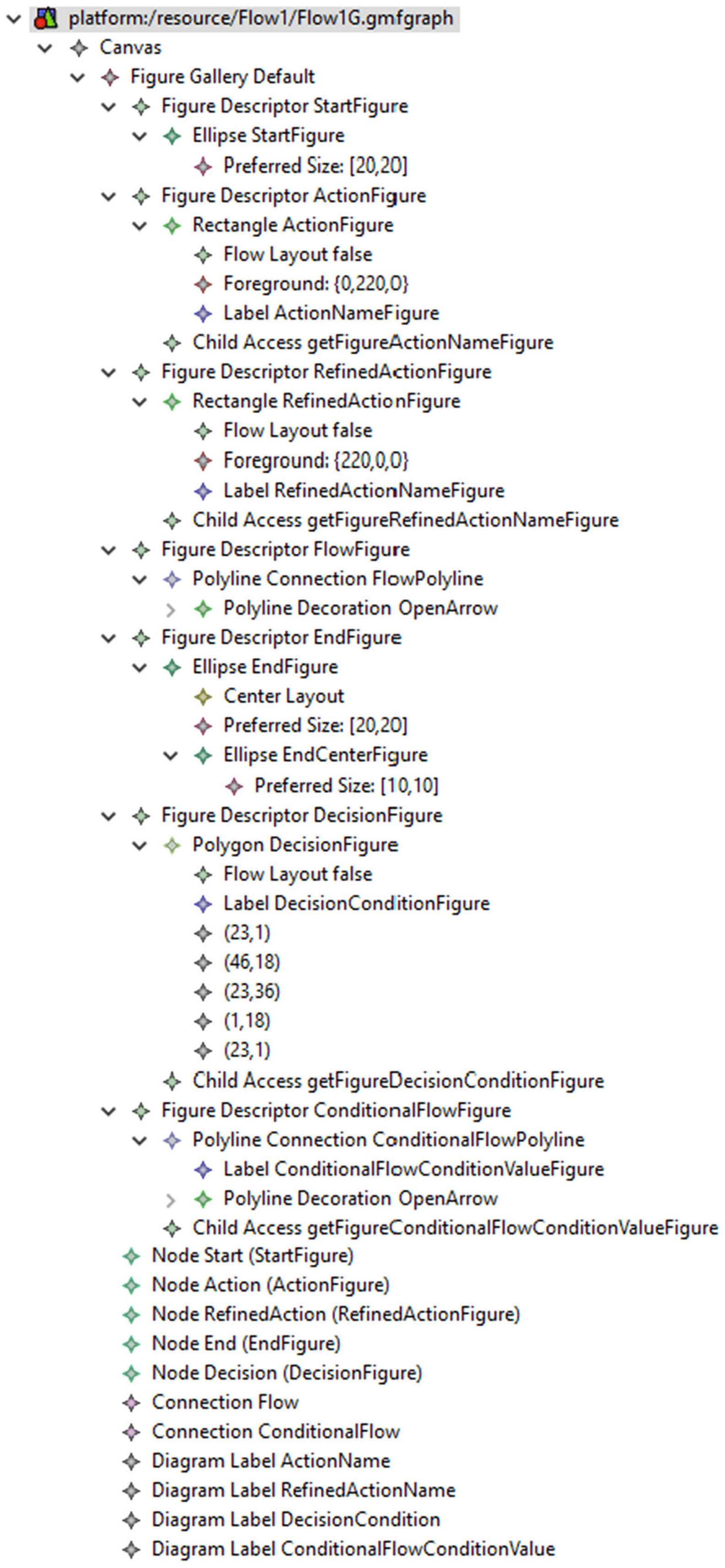
Figure 4 shows the tooling model. Only creation of nodes and edges is present in the palette.

Figure 5 shows the mapping model for the Flowchart. Again, the tree view of the model does not reveal all relevant aspects, and some important features are visible only in property dialogs, e.g., what model element in fact is represented by a string in a mapping text. The initial part of the mapping model shows diagram node mappings. The first is the decision node, the Top Node Reference line for it shows that the Decision in the domain model is reachable from the top class Flowchart representing the diagram itself via the path *Flowchart.flowEnd* (in fact, it leads to the superclass FlowEndNode of the Decision class). The node mapping line shows that the Decision class in the domain is mapped to the decision node in graphics. The Feature Label mapping shows that the *condition* attribute of Decision in the domain is mapped to the DecisionCondition label in graphics. Mapping details for other nodes are similar. Since edges in our domain are modeled as classes (the alternative could be just associations), for them the links to start and end classes in the domain are explicitly defined. If required, OCL constraints can be used for classes—here only one constraint is required specifying that the Action subclass RefinedAction is not to be mapped as the Action itself. The mapping definition concludes with references to the other GMF models used.

We do not show here the generation model since it contains only technical information. Thus, the flowchart graphical syntax (to a degree) and editor functionality (except the property editor for attributes which is not visible in a model) is shown. We see that even for this simple language the definition is not trivial.

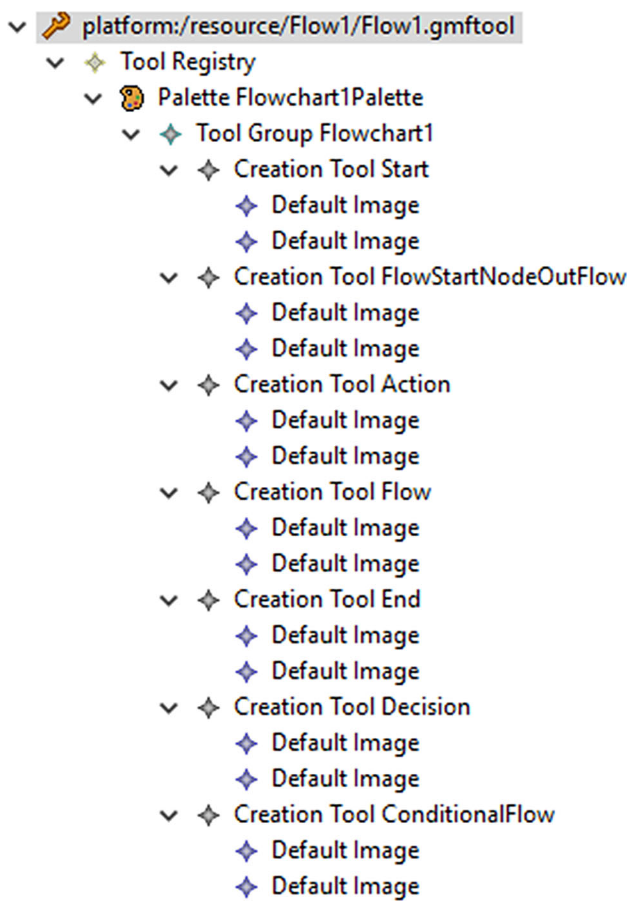
Several research teams have come to the conclusion that though the GMF approach is quite universal for graphical editor developments, in practice its application is quite difficult, especially for more complicated graphical languages. Therefore, several trials to simplify the usage of GMF have been proposed. The first one is the Obeo designer [10,11] where a graphical diagram is defined as a viewpoint of the domain model. In paper [11], the Obeo team notes that while the creation of an ER-style graphical editor for practical database analysis would require 30 days for a GMF expert, it could be done in 5 days by domain expert in Obeo designer. Now a version of Obeo designer named Eclipse Sirius [12] has appeared, but the main features for developing graphical tools are the same. The main development step there is the definition of the viewpoint of the domain model which specifies the diagram description. This description combines the chosen set of graphical elements for the diagram with their mapping to the corresponding domain model elements (thus, in fact, GMF graphical and mapping models are merged here, but in an alternative notation). Graphical elements include various kinds of containers which are equivalents to GMF nodes with compartments and compartments themselves. The nodes here are simple ones containing only texts and/or icons. Graphical style for a node supports the shape (a richer set than in GMF), color and size. Bordered nodes are also offered for representing ports. Edges represent connections of elements, and they can be mapped to either domain associations or classes. The mapping definition can be extended by expressions in Acceleo [13], which is a superset of OCL, or in pure OCL. Editor functionality is defined via tools, which are associated with mappings. A tool can have an applicability condition expression and the body describing its action in a special Operations language. Operations include also invocation of a property dialog for editing element properties. An operation can invoke also external Java code. Thus, the Sirius approach seems to be sufficient for creating complicated graphical tools, but its ease of usage is not so clear.

Another approach which criticizes GMF for complexity and tries to simplify it is Eugenia [14,15]. Especially in [14], the insufficient quality of GMF wizards offering initial versions of graphical, mapping and tooling models from the domain model is criticized, since in the result all these three models in fact have to be built manually from scratch. It should be noted that such an effect occurred also for the Flowchart example described above. Authors of Eugenia try



```
platform:/resource/Flow1/Flow1G.gmfgraph
  Canvas
    Figure Gallery Default
      Figure Descriptor StartFigure
        Ellipse StartFigure
          Preferred Size: [20,20]
      Figure Descriptor ActionFigure
        Rectangle ActionFigure
          Flow Layout false
          Foreground: {0,220,0}
          Label ActionNameFigure
        Child Access getFigureActionNameFigure
      Figure Descriptor RefinedActionFigure
        Rectangle RefinedActionFigure
          Flow Layout false
          Foreground: {220,0,0}
          Label RefinedActionNameFigure
        Child Access getFigureRefinedActionNameFigure
      Figure Descriptor FlowFigure
        Polyline Connection FlowPolyline
          Polyline Decoration OpenArrow
      Figure Descriptor EndFigure
        Ellipse EndFigure
          Center Layout
          Preferred Size: [20,20]
          Ellipse EndCenterFigure
            Preferred Size: [10,10]
      Figure Descriptor DecisionFigure
        Polygon DecisionFigure
          Flow Layout false
          Label DecisionConditionFigure
          (23,1)
          (46,18)
          (23,36)
          (1,18)
          (23,1)
        Child Access getFigureDecisionConditionFigure
      Figure Descriptor ConditionalFlowFigure
        Polyline Connection ConditionalFlowPolyline
          Label ConditionalFlowConditionValueFigure
          Polyline Decoration OpenArrow
        Child Access getFigureConditionalFlowConditionValueFigure
    Node Start (StartFigure)
    Node Action (ActionFigure)
    Node RefinedAction (RefinedActionFigure)
    Node End (EndFigure)
    Node Decision (DecisionFigure)
    Connection Flow
    Connection ConditionalFlow
    Diagram Label ActionName
    Diagram Label RefinedActionName
    Diagram Label DecisionCondition
    Diagram Label ConditionalFlowConditionValue
```

Fig. 3 Graphics model of the flowchart



```
platform:/resource/Flow1/Flow1.gmftool
  Tool Registry
    Palette Flowchart1Palette
      Tool Group Flowchart1
        Creation Tool Start
          Default Image
          Default Image
        Creation Tool FlowStartNodeOutFlow
          Default Image
          Default Image
        Creation Tool Action
          Default Image
          Default Image
        Creation Tool Flow
          Default Image
          Default Image
        Creation Tool End
          Default Image
          Default Image
        Creation Tool Decision
          Default Image
          Default Image
        Creation Tool ConditionalFlow
          Default Image
          Default Image
```

Fig. 4 The tooling model

to improve the usability by offering high-level annotations to domain model elements and using model transformations to generate the required GMF models automatically. This permits also to synchronize easily the true definition sources (domain model + annotations) after some modifications with the other models. The annotation for a domain model class defines the GMF node by which it should be visualized, including figure, color, border style, the contained labels for class attributes, and tool features; thus, everything in the graphic, mapping and tooling models for this class in fact is specified. Similar principles are used for other domain model elements. There is a standard transformation in Epsilon transformation language [16], which generates other models automatically. If required, the tool definer can add a custom transformation polishing some details of the models. Finally, a standard Epsilon transformation builds the generation model from the previous models, and from it the tool is obtained by the standard GMF generator. The papers [15,17] report some results of more direct comparison of usability of GMF to the usability of its improvements. Thus, the experimental development of a graphical editor for BPMN metamodel took 25 days for pure GMF, 5 days for Obeo designer (Sirius) and about 3 hours for Eugenia (by medium qualified developers).

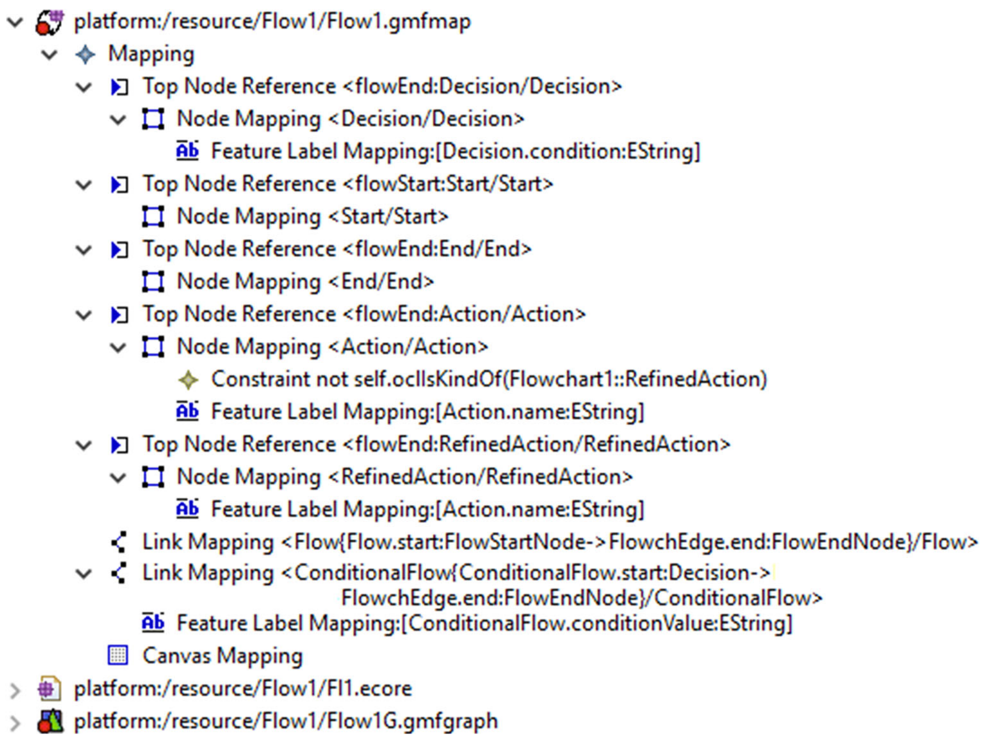
There is also a paper [18] reporting some experience of using GMF for the development of graphical DSLs in industry, namely the Siemens Company for development of embedded software. The conclusion is similar—the derivation of other required models from the domain model is error-prone and should be performed in minimal steps with immediate testing. Thus, despite of some success stories of serious practical usage of GMF, e.g., as the basis for the well-known IBM UML modeling tool RSA [19], improvements of the GMF technology are really required, in order to support wide graphical DSL building.

Finally, the IMCS UL Modeling team, including the authors of this paper, has also some experience in using GMF for graphical DSL and their tool development. Initially, an attempt was made to use GMF as the basis for the whole IMCS graphical DSL project. This research resulted in the development of METAclipse—an extension of GMF framework for implementing graphical model transformation languages [20]. Several shortcomings of GMF were found, e.g., the fact that the graphical model in GMF has no instances at runtime (a hidden Notation model is used instead). These shortcomings were neutralized by direct extension of GMF at Java code level. In the result, a very flexible relation between the domain model and graphical presentation model of the language was achieved in METAclipse by using model transformation languages, in the given case MOLA [21]. The details of METAclipse platform and its implementation are published in [22]. METAclipse is still used today for the MOLA tool [23]. In this tool, the graphical editor for MOLA is closely integrated with the language syntax checker and compiler.

Despite this success, for the general modeling language project at IMCS UL the METAclipse approach was not used, because in this project the main emphasis from the very beginning was on defining DSLs directly via their graphical syntax. In addition, this approach still was considered to be too complicated for simple graphical modeling languages. In the result, the TDA tool definition platform, to be discussed in the next section, was developed.

3 Graphical language definition in TDA platform

In this section, our TDA tool definition platform [6,7,24] developed at IMCS UL is briefly discussed. The main design goal of this platform has been to provide a simple environment for defining graphical modeling languages and their tools directly on the basis of their graphical syntax. The platform is fully based on metamodel instantiation. A simple and natural Type metamodel for graph diagrams is proposed.



```
platform:/resource/Flow1/Flow1.gmfmap
  Mapping
    Top Node Reference <flowEnd:Decision/Decision>
      Node Mapping <Decision/Decision>
        Feature Label Mapping:[Decision.condition:EString]
    Top Node Reference <flowStart:Start/Start>
      Node Mapping <Start/Start>
    Top Node Reference <flowEnd:End/End>
      Node Mapping <End/End>
    Top Node Reference <flowEnd:Action/Action>
      Node Mapping <Action/Action>
        Constraint not self.oclIsKindOf(Flowchart1::RefinedAction)
        Feature Label Mapping:[Action.name:EString]
    Top Node Reference <flowEnd:RefinedAction/RefinedAction>
      Node Mapping <RefinedAction/RefinedAction>
        Feature Label Mapping:[Action.name:EString]
    Link Mapping <Flow{Flow.start:FlowStartNode->FlowchEdge.end:FlowEndNode}/Flow>
    Link Mapping <ConditionalFlow{ConditionalFlow.start:Decision->
                  FlowchEdge.end:FlowEndNode}/ConditionalFlow>
      Feature Label Mapping:[ConditionalFlow.conditionValue:EString]
    Canvas Mapping
platform:/resource/Flow1/Fl1.ecore
platform:/resource/Flow1/Flow1G.gmfgraph
```

Fig. 5 The mapping model

The graphical syntax of the given language is defined as an instance of this metamodel. There is also an extended version of this type metamodel. An instance of this metamodel directly defines a graphical editor for the language—this instance can be interpreted by the presentation engines of the platform. To keep all this simple, the Type metamodel is in fact at MOF level M1—a class diagram, but its instance is at M0—an object diagram.

We start with a simplified version of Type metamodel in TDA (Fig. 6). It is a fixed metamodel which contains type classes for all elements of a graphical diagram language—GraphDiagram, Node, Edge and Compartment (of Node or Edge). It should be noted that the terminology used here slightly differs from that used in OMG DD and Eclipse GMF—in our approach, a Compartment is any textual element in a diagram, single line or multiline—both the Compartment and Label in the sense of OMG DD [3] or GMF. In DD, a Compartment is a part of Node delimited by horizontal lines which can contain many text lines (Labels) or other Nodes. Our Compartment contains only text—Node nesting is specified in another way. Classes of the Type metamodel contain the basic style attributes of diagram elements—those which typically are fixed when a diagram syntax is defined informally, such as node shape and line-end shape. For the sake of simplicity, other inessential style attributes such as color and font size are not included. A node compartment may be a structured text, e.g., a text line for a class attribute consists of attribute name, type, initial value, with relevant

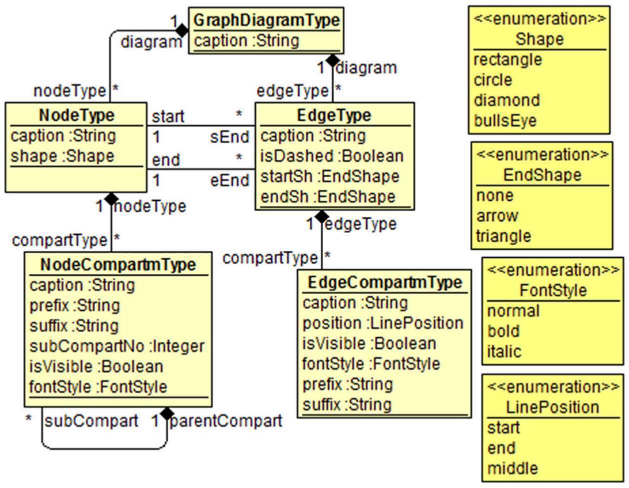


Fig. 6 Simplified type metamodel

separator strings included. This structuring is enabled by the *parentCompart—subCompart* association and *prefix*, *suffix* and *subCompartNo* attributes. It should be noted that in practical diagram definitions (including OMG standards) these aspects typically are defined by including context-free grammar fragments. The concepts of our Type metamodel have been chosen with the goal to simplify diagram editor definition in TDA.

We remind that the syntax for a concrete diagram notation is defined as an instance of this model (a UML object dia-

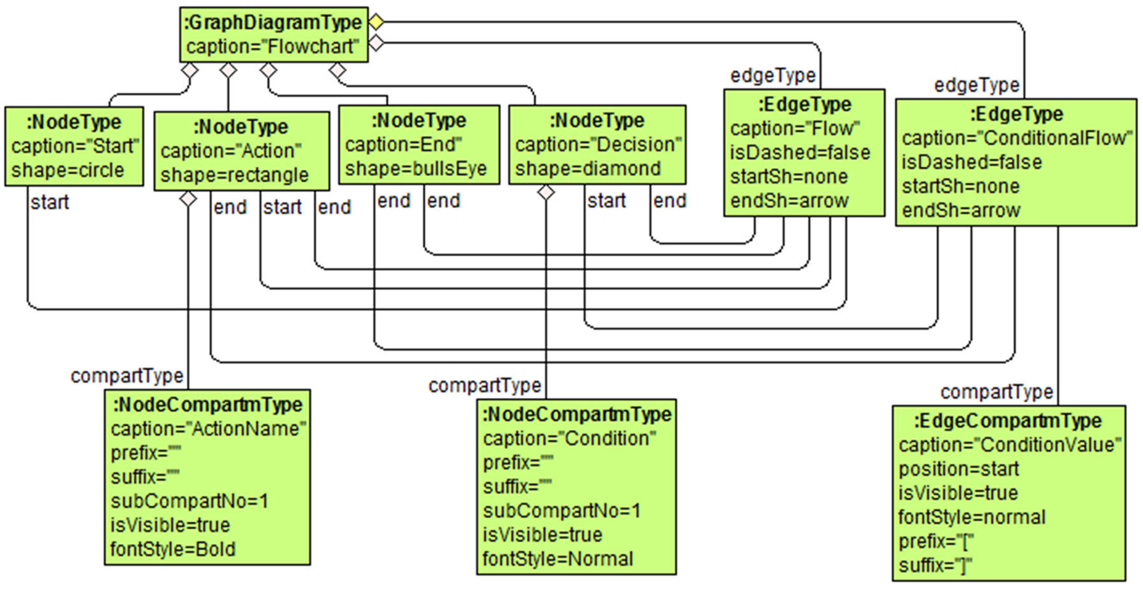


Fig. 7 Flowchart syntax definition by instantiation (simplified)

gram). The semantics of this definition is slightly implicit—it is assumed that each node in a diagram has one of the defined node types, each edge—one of the edge types and so on.

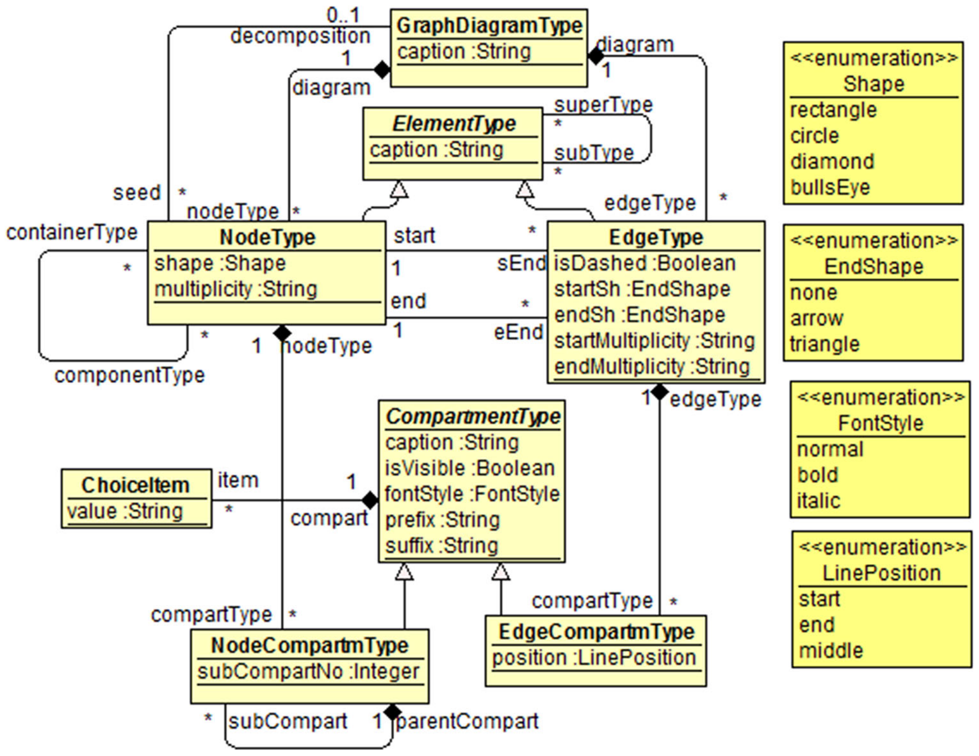
Figure 7 shows an example of flowchart syntax definition obtained this way. This version is slightly simplified with respect to the version provided in Section 2. There are only four node types—start node, action node, decision node and end node. We would like to retain also the restrictions—e.g., that 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. However, these constraints cannot be specified in Fig. 7. Only the basic structure of the intended flowchart syntax can be defined by the object diagram in Fig. 7—which edges can start from which nodes, what texts are associated with the diagram elements. But the element multiplicities cannot be defined this way—multiplicity constraints cannot be attached to links in a UML object diagram. In addition, no true OCL constraints can be attached to elements of syntax definition—constraints can be added only to UML classes.

In the original TDA platform [7], multiplicities and other missing features are introduced in a custom way. Some specific attributes and associations are added to the type metamodel, which permit to add the missing functionality to an instance of this type metamodel by creating links and slots for these specific metamodel elements. Then, they are interpreted in a custom way to simulate multiplicities, generalization and simple constraints otherwise not available in object diagrams.

Figure 8 shows this enhanced Type metamodel in TDA. The multiplicity for edges is introduced by custom attributes *startMultiplicity* and *endMultiplicity* for the EdgeType class. Their values specify accordingly how many edges of this type can maximally start from the node type referenced by the *start* link and how many can end into the node type referenced by *end* link. If no value is specified in a type instance, then no restrictions apply, i.e., * multiplicity in UML terms. Certainly, in this specification it is assumed that no more than one *start* or *end* link exits from an EdgeType instance. The next custom element is the class specialization simulation by *subtype*/*supertype* association in the type metamodel. It can be applied to both node and edge types. This subclass simulation is semantically complete—all slots and links from the supertype instance are assumed to be relevant also for the subtype. The complete Flowchart syntax definition in Fig. 9 shows how this imitated subclass feature permits to avoid repeating of *start* or *end* links to several instances, and common slots also need not to be repeated for subtypes (in fact, there are “surrogate superclasses”— FlowStart et al for the true node type instances). The *caption* attribute for all classes in fact simulates the class instance name (informally—the diagram element name the instance represents). One more specific feature is the ChoiceItem class. Its instances permit to define the possible value set for a compartment type. Thus, a simple OCL constraint saying that only a constant set of values is valid for the given compartment can be imitated. However, the case when this set is variable—dependent on other diagram elements, cannot be imitated this way.

Figure 9 shows the complete flowchart syntax definition as an instance of the complete Type metamodel in Fig. 8. All features and constraints from the informal Flowchart definition in Sect. 2 are included in this instance (except for uniqueness of action names and uniqueness of conditional flow labels from a decision). Thus, this approach is more or less sufficient for not very complicated graphical language

Fig. 8 Complete type metamodel in TDA



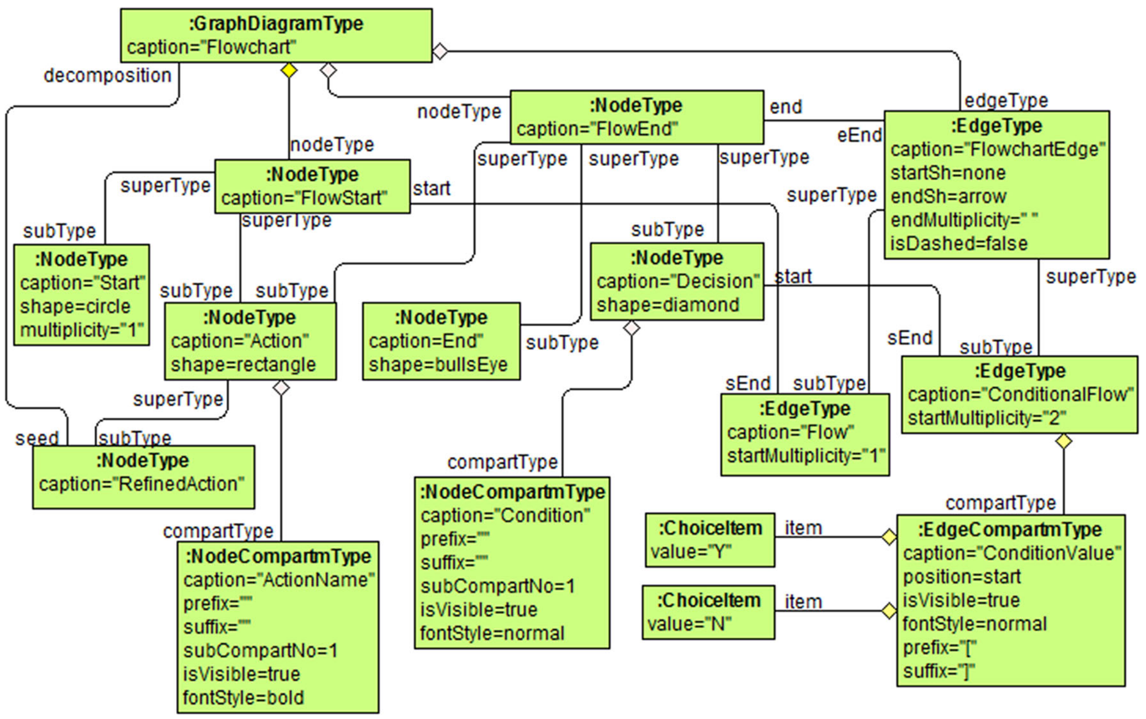


Fig. 9 Complete definition of flowchart language by type metamodel instantiation

definitions. Only the readability of these definitions with the nonstandard interpretation of object diagrams is not so clear for a non-experienced reader.

Further extension of Type metamodel in TDA is used for graphical editor definition for a graphical language set. All language definition features are retained, but a number of new editor-related classes are added—extended style elements, palette, menus, text editing options, etc. Especially, this metamodel now contains both diagram elements and their types, linked by an association. Namely, an instance of this

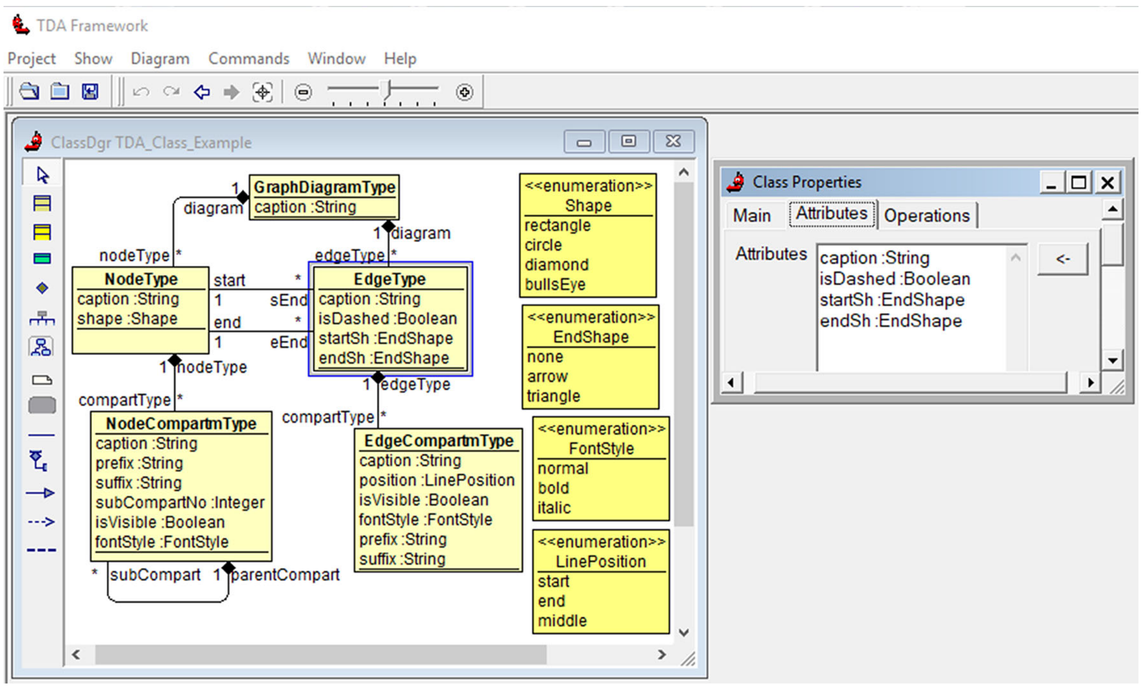


Fig. 10 Screenshot of class diagram editor built using TDA framework

metamodel directly defines an editor—it can be directly interpreted by the engines in the TDA framework. The principles of type metamodels in TDA to a large degree are influenced by the desire to simplify the interfaces of these engines (they are also defined as metamodels). In this context, the nonstandard semantics of some model elements is fully acceptable. In the editor context, the issue of missing dynamic constraints is fully solved—a dynamic solution is proposed. The editor metamodel contains the concept of extension points in the editor functionality. At these extension points, custom model transformations in Lua/lQuery language [25] can be invoked, thus extending the standard behavior of TDA engines. There is a Configurator tool [26,27] in the platform which helps to build an instance of the Type metamodel in an easy way. All this together makes the TDA platform usable in practice for the definition of nearly any graphical DSL and its support tools. However, then a significant knowledge of TDA model and engine internals is required from developers.

The TDA platform has been successfully used for several large practical projects. The most notable one is the OWLGred editor [28–31] built by IMCS UL for visualizing OWL ontologies in a class diagram-like notation. This tool is freely available in the web [32], and it can import various OWL textual notations and has been used by many ontology development teams in the world. Other TDA applications include the development of custom graphical workflow notations and tools for several companies and institutions in Latvia [33]. However, this second application shows also the complexity of full TDA approach, since it is difficult for business analysts to maintain and extend the tools. This complexity aspect has been the main stimulus for the development of metamodel specialization approach—the main topic of this paper. One more application of TDA is the free simple UML tool GradeTwo [34]. This tool is widely used at the University of Latvia for teaching UML basics. By the way, all class diagram examples in this paper are built using this tool. Figure 10 shows an example of screenshot of the class diagram editor in GradeTwo built using the TDA framework, and this example contains the same class diagram as in Fig. 6.

4 Diagram definition standard proposed by OMG

In this section, we provide a very brief overview of the new OMG standard for Diagram Definition (DD) [3]. The first version of this standard appeared in 2012, and the current version is from 2015. As already stated in Introduction, for a long time the graphical syntax (notation) of all OMG standard modeling languages was specified informally. Only starting from UML 2.5 [2], the graphical syntax is defined using the OMG Diagram Definition (DD) standard [3]. DD standard, in turn, consists of Diagram Interchange (DI) and Diagram Graphics (DG) parts, each having a metamodel. The official intention of the DI part is to specify at a logical level those aspects of a diagrammatic language (e.g., UML) which are vital for correct diagram interchange between different tools supporting this language. Therefore, some structure of diagram elements in the language is defined in DI. The graphical form of the language elements is defined only by a transformation-based mapping of DI elements to DG elements. The DI metamodel contains no concepts such as rectangle or line, and they are present only in the DG. It is emphasized that those graphical aspects of language elements which are strictly defined in the given language standard are not included in DI, because they are common to all tools for this language. In addition, in the DI metamodel the diagram structure is defined at a very high abstraction level. It is tried to give some sense to diagram elements by having a relation from a DI element to an element it can represent in the metamodel of the abstract syntax (domain model) of the language. To add more directly some meaning to elements of the DI metamodel which describes diagrams in a language, it is proposed to specialize the DI metamodel to an interchange metamodel for the language, with subclasses defining more specific language elements. The specialized UMLDI metamodel is given as an example in the OMG standards [2,3]. In UML 2.5 [2], the UMLDI metamodel covers a relatively larger part of UML, but no mapping to DG is present. In the DD standard [3], only the class diagram related part is present in UMLDI, and a mapping to DG is shown. 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 (the difference is that an association subsetting is also used, since the original associations are significantly more abstract, new attributes can be added to subclasses as well). The DI metamodel bears some similarity to the UMM for diagram definition (see Fig. 15) used here (Shape is similar to Node, Edge is used in both). However, the abstraction level is much higher, e.g., no explicit concept of textual element is in DI. The provided specialization, UMLDI, is also more abstract than we would use, e.g., for Class diagram definition (see Sect. 10). This high abstraction approach leads to the fact that an overwhelming part of the concrete UML diagram structure in UML 2.5 is defined informally as comments to UMLDI.

The DG metamodel contains purely graphical diagram elements—rectangle, circle, line, etc., with style attributes relevant for rendering an imported diagram. The diagram elements in this metamodel contain no hints at all on their expected usage in a diagram; instead, they have a similarity to the elements of the SVG [35] rendering standard (in some sense also to GEF mentioned in Sect. 2). It is assumed that elements of the specialized DI metamodel are mapped to elements of DG using some model transformation language; e.g., in [3], the MOF QVT [36] operational mappings are used. Therefore, the concrete graphical syntax of the language can only be understood by reading this transformation.

The same way as for other existing approaches, we try to demonstrate the DD approach on our Flowchart example. We start with a possible version of Flowchart DI metamodel. This version is built using the UMLDI in [2] as an example, but with slightly more details. The domain model of flowchart created for the GMF approach (see Fig. 2) is interpreted here as a domain metamodel and is also extended by a "top superclass" FlElement—the usage of this model as a reference for Flowchart DI requires such superclass (an equivalent to the Element in UML). The flowchart domain (meta) model is assumed to be in the package named Fl. Figure 11 shows this Flowchart DI metamodel. Only the start, end and action nodes and flow edge are present there because the goal is simply to illustrate the situation. But other nodes and edges could be added here in a similar way (nodes are renamed to shapes according to DI traditions). In fact, all basic features (edge multiplicity constraints, labels inside nodes) are already demonstrated here.

The next step according to the DD methodology would be to define a model transformation from Flowchart DI model to the DG model (in any appropriate transformation language—MOF QVT, ATL, etc.). We will not show this step here, because the UML class diagram example from the DD standard [2] and paper [37] shows that an easy readable mapping from the specialized DI metamodel to DG metamodel cannot be defined even for simple situations. And without this mapping, there is no hint at all in Fig. 11 how the graphical form of a flowchart should look like. Simple style attributes which could be added to DI elements do not include the real graphical form of a shape—is it a rectangle or circle. Thus, the goal of defining a simple and easy readable formal specification of the graphical syntax for a diagrammatic language most probably cannot be reached this way. This is confirmed by the fact that in UML 2.5 [2] the graphical syntax is still defined informally on examples, and only tables relating these example fragments with corresponding UMLDI classes are added. At the same time, the real practical goal of the DD standard—to define a diagram interchange between tools implementing the same graphical language, can be reached; experiments with UML tools in [37] confirm this.

An interesting issue is whether the DD approach can be used for the second topic of this paper—graphical editor definition. At a first glance, it seems that the approach provides no facilities in this direction. However, there has been one partially successful experiment in this direction. In Models 2015 paper [38], the authors of the DD approach investigate on the basis of their previous experiment with UML class diagrams [37], whether a UML class diagram editor could be built in a standard way on the basis of this diagram syntax defined via DD. Eclipse GMF is chosen as the target environment. The first result is very interesting—a considerably more detailed UMLDI specialization is required, in some sense similar to that we propose to use in Universal Metamodel specialization for class diagram editor definition in Sect. 10. However, there are several problems in the DD approach. The first one is to find the correct correspondence between the associations in the specialized DI metamodel and their counterparts in the domain metamodel—only classes of the specialized

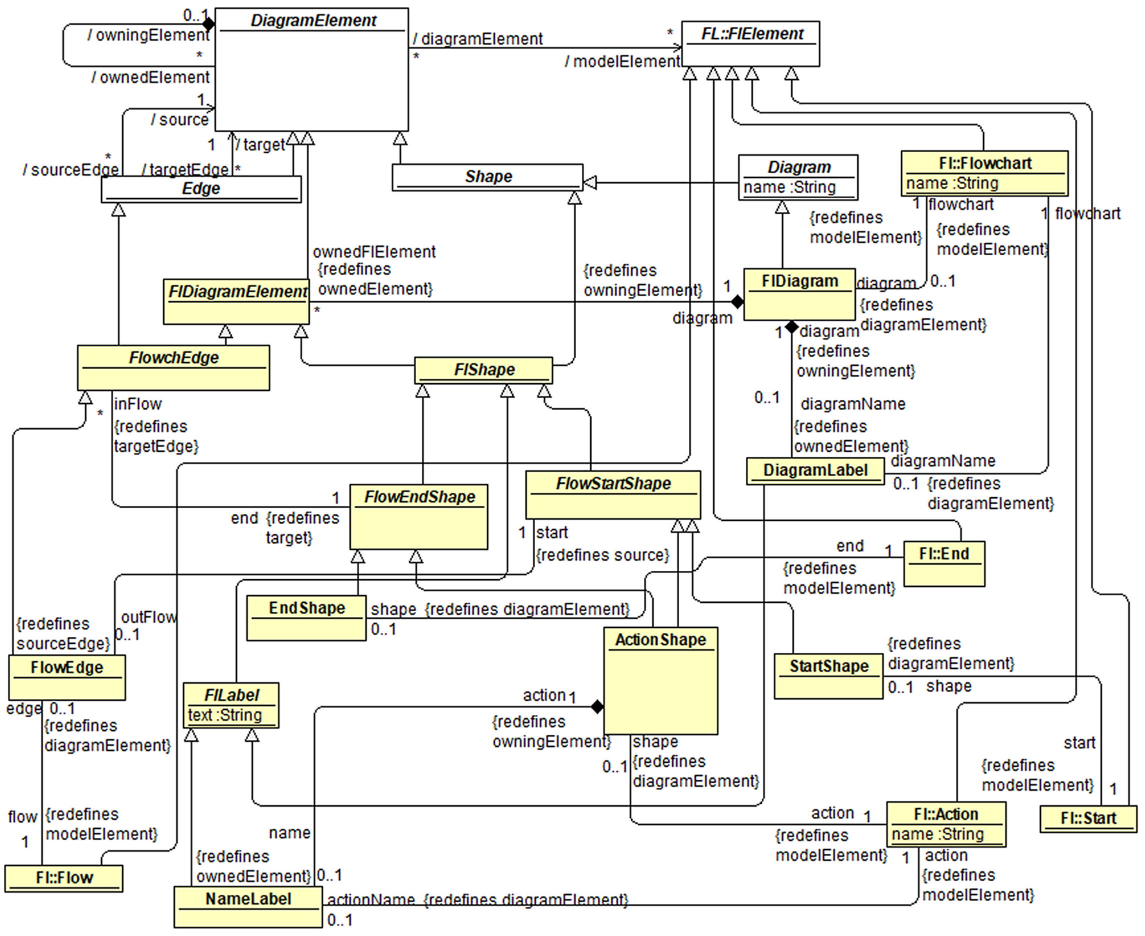


Fig. 11 Flowchart DI metamodel (fragment)

DI are directly related to domain classes; for associations, an ambiguity can appear. Care should be taken to relate correctly the responses to graphical user actions (strictly bound to graphical objects) with diagram logical structure modification (related to DI). Finally, one more problem appears in the implementation, because the desire is to obtain directly the GMF Generation model. This goal requires a deep understanding of the GMF Generation model internals and even more deeper specialization of UMLDI metamodel (provided by using stereotypes for UMLDI classes). Formally, the experiment in [38] was successful—a GMF based editor for the class diagram subset was generated from the provided DD. But without further extensions of the DD facilities the practical generation of usable editors from language definition via DD seems to be not so easy.

5 Other existing approaches to graphical syntax and editor definition

We start with the graphical syntax topic. Besides the metamodel based DD approach for graphical syntax definition, there are also some approaches based on other formalisms such as graph grammars, including eXtended Positional Grammars [39] or Layered Graph Grammars [40]. But none of them seems to be as usable as the metamodel based approach. Evidently, there is no active research in this direction at a time.

Now about platforms for graphical editor definition. There are a lot of such platforms, not only the Eclipse GMF and TDA already discussed. Most of them are based on the abstract syntax (domain metamodel) definition of the language via the classic MOF approach and adding some sort of mapping from this syntax to graphical notation elements. There are more Eclipse-based variations of GMF than the two already mentioned. There is the Tiger platform [41,42] adding the possibilities of the graph transformation language AGG [43]. The original version of Tiger [41] used graph transformations for linking the domain model in EMF to GEF based editing of diagrams, but a newer version [42] offers graph transformations of the domain model for defining extended editing commands in addition to basic editing offered by GMF. A similar approach is the ViatraDSM platform [44] which defines a mapping from the domain model to GEF-level presentation concepts using the Viatra2 model transformation language [45]. An alternative to GMF is the Graphiti framework [46] which is more Java programming than model oriented. In addition to a typical domain model in EMF, an alternative Pictogram model is used for graphics definition and their linking via a Link model (which must be implemented via a Diagram type agent). There is a rich program library based on simple APIs for building such agent; similarly, the diagram rendering is provided by an extendable internal rendering engine (which in fact is based on GEF runtime). There is now also an offspring of the described GMF extension Eugenia—Eugenia Live [47]. This platform is especially aimed at graphical DSL design and experimenting with them. Originally, there is no fixed metamodel of the new DSL; instead, a diagram example and its future metamodel can be built in parallel. The metamodel of the platform itself contains both Node and NodeType classes linked by an association, similarly for edges, etc., and there is no true domain model at all. Thus, the approach has some similarity to TDA at metamodel level, and an interpreter-based execution is used in both. The main difference is in the usage style. In TDA, the type instances have to be explicitly defined using the Configurator [27] before the diagram building. But in Eugenia Live the type instances constituting the palette (a sort of future definition of the language) are created on the fly, the same way as elements of the diagram example; however, a type instance must be created before using it as a palette element for node. The platform is implemented as a browser plug-in in JavaScript-related [48] languages. When the experiments with a new DSL are complete, the result can be exported for future use in Eclipse GMF (Eugenia version). The approach is promising, but currently only the basic diagram features (when compared to TDA) are implemented.

A special position in the list of Eclipse related platforms is taken by Collaboro platform [49], aimed at collective development of a DSL by a team. Therefore, the support of team development there comes at foreground, even before the language definition facilities themselves. The language definition is based on metamodels in a classical way—here is a Domain metamodel in EMF and a Notation (concrete syntax) metamodel. The Notation metamodel contains elements for graphical and textual syntax simultaneously; thus, a textual and graphical syntax for a DSL can be developed in parallel. Currently, the language is defined by instantiation of both metamodels, with a notation instance having a link to the corresponding domain instance (a static mapping). The team development process itself (proposal, versions, voting, etc.) is defined by a metamodel as well. The platform is implemented via an Eclipse-based backend on a server and a frontend to be used by each team member—either as a browser plug-in or a front end in Eclipse. The backend supports both the language to be developed and the team development process.

Certainly, there are also many non-Eclipse-based graphical tool definition platforms. Microsoft DSL [50] uses a "standard" pattern by starting with a domain metamodel and then adding the presentation and mapping metamodels and ending up in code generation, and only metamodels are created in a "dialect" of UML. A completely domain-specific metamodeling language GOPPRR is used in the MetaEdit [51] platform where the graphical syntax metamodel can be defined directly, but with limited functionality, and still involving some code generation. A common feature for all these platforms, and some similar ones, is that for each new DSL a new metamodel must be created in some metamodeling language. There is also an early approach (2003) for creating graphical modeling tools on the basis of graphics metamodels (typed graphs) and graph transformations— the GenGED platform [52]. The metamodel is used to define the graphical elements of the language. The graph transformations must be defined in the AGG language [43] to specify the permitted editor behavior for creating syntactically correct diagrams (a sort of executable grammar rules). A similar approach is used also in DiaGen [53] and ATOM3 [54] platforms.

6 New approach to graphical language and tool definition based on metamodel specialization

The previous sections devoted to analysis of the existing approaches to language and tool definition based on instantiation have indicated a number of shortcomings and problems, especially for language graphical syntax definition, where the OMG DD approach has many problems. Our goal in this paper is to offer a new approach based on metamodel specialization which is superior in a number of cases.

Class specialization by 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 for 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.

Now let us explore this idea in details.

The starting point for the application of the specialization approach is to define the metamodel to be specified—the Universal Metamodel (UMM) for the given task. Certainly, this makes sense in situations when there are many similar cases of the task, each of which can be more or less completely defined by a specialization of the UMM. A very appropriate example of such task is the graphical syntax definition of a node–edge style graphical language, to be considered in the next section. But there are other similar tasks—they will be considered in subsequent sections. The most important

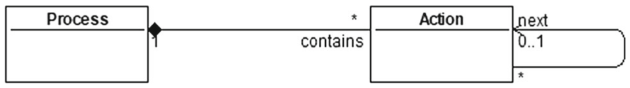


Fig. 12 UMM for a simple workflow

of them is the graphical editor definition for the language. It should be emphasized that the Universal Metamodel for this task is a direct extension of the UMM for language syntax definition.

The metamodel specialization is based on standard UML features—creating subclasses of the base metamodel (UMM), redefining (or subsetting) class properties and adding new OCL constraints. To make the notation simpler and more compact, we slightly restrict the metamodel specialization facilities to be used. Any number of subclasses may be defined for a UMM class. For subclasses, only the attributes inherited from UMM may be used—they are redefined by default. Thus, we can add new default values, but we do not redefine attribute names, types and multiplicity. Associations between subclasses must redefine the corresponding associations from UMM (explicit redefinition must be used when new role names are introduced), and subsetting is not used. Thus, no new attributes or associations may be introduced in the specialization. Arbitrary OCL constraints may be added to classes and attributes in the specialization. A specialized class may be also abstract, with concrete subclasses in the specialization. We remind that the defined specialization of UMM remains at the same MOF layer as the UMM itself.

We illustrate the approach on a very simple example—a custom workflow language definition. Figure 12 shows the UMM for this language – it contains just two classes with no attributes. A specialization of such generic workflow could, for example, define a Business Trip or a document submission to a state institution. For each of such workflow cases, custom names of actions and a specific permitted sequence of actions could be defined. Figure 13 shows the specialization defining a Business Trip.

The specialization still represents a metamodel for Business Trip. Classes of the specialization can have instances in a normal way, e.g., TripToBerlin, ReserveLufthansaFlight. The explicit *redefines* modifier is used here only for the redefined role names. Classes of a UMM will be shown in this paper with a white background, but the specialized classes—with a colored one.

In order to make the metamodel specialization examples more readable and compact, in this paper we use a custom notation for specialized classes and redefined associations—we show only the specialized classes and add the original class and role names from UMM in braces (and in bold italic font), and the redefinition is shown without the *redefines* keyword (in a similar style). See Fig. 14 which presents the same specialization as in Fig. 13.

Finally, some comments on other usage cases of metamodel specialization. Despite the fact that class specialization is a well-known UML feature, there are very few usage cases of the whole metamodel specialization. The closest one to our approach is the usage for DI metamodel specialization [2,3,38], e.g., UMLDI, already mentioned in Sect. 4. In particular, in [38] the usage of specialization for defining an appropriate level of details for UMLDI is explained in details and the conclusion is that the approach is adequate for the goal. Some other cases of metamodel specialization ([55,56]) are completely unrelated to the topic of this paper. One more aspect to be mentioned here is that the most used metamodeling paradigm—instantiation according to the multi-layer approach in MOF—has its internal problems as well. As pointed out by several researchers—T. Kuhne and C. Atkinson [57,58] and B. Henderson-Sellers [59], MOF instantiation approach is not very precise from the formal semantics point of view. Various improvements such as strict versus loose metamodeling [60,61] and a Metamodeling Kernel [62] have been proposed. All this could mean that the metamodel specialization paradigm is also more precise formally than the traditional instantiation. It should be noted that the above research on instantiation has been done in the context of a broader topic—multilevel modeling

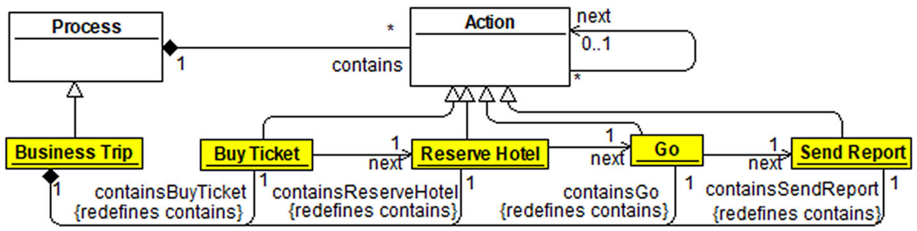


Fig. 13 Workflow specialization defining a Business Trip

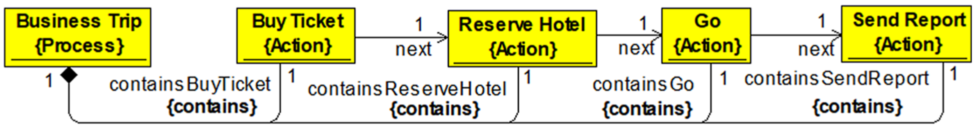


Fig. 14 Custom notation for the Business Trip specialization

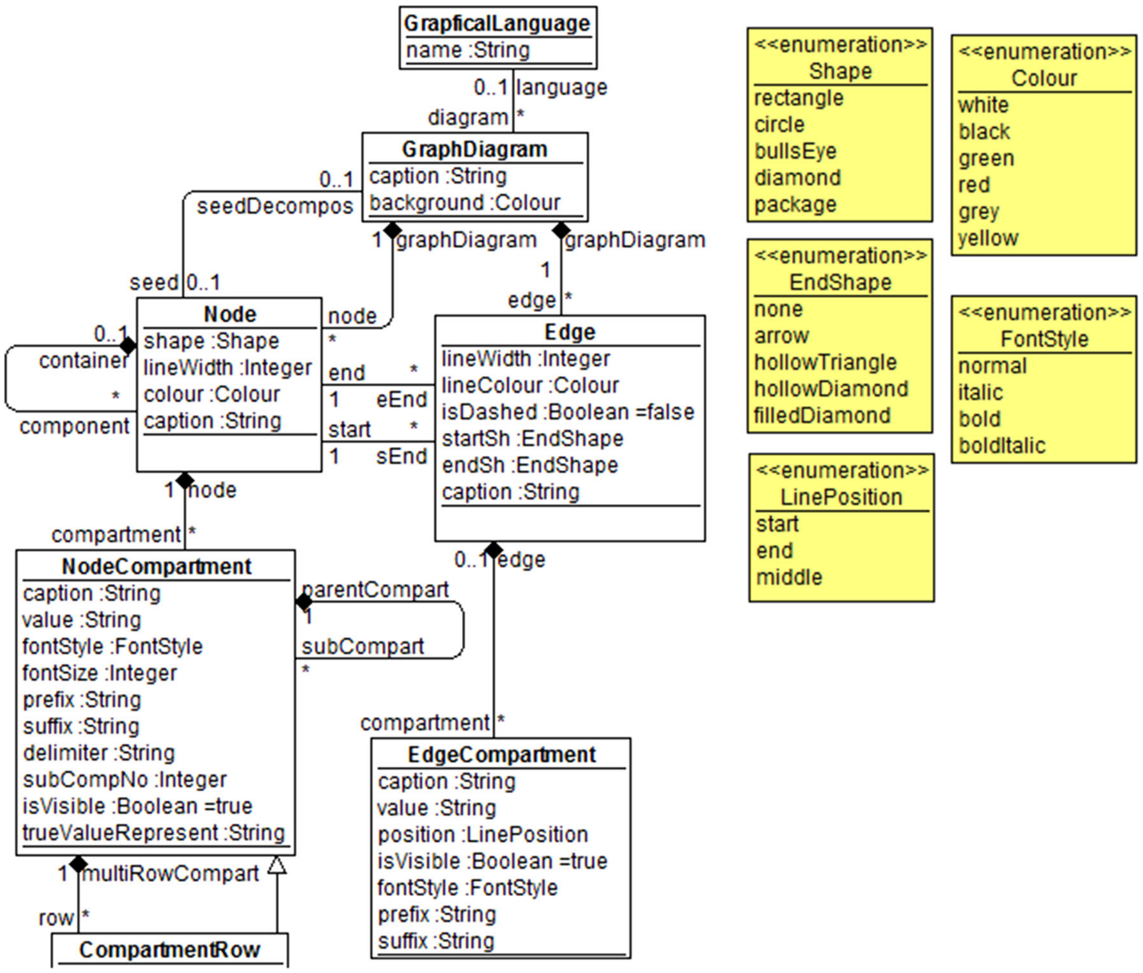


Fig. 15 UMM for diagram syntax definition

(where unlimited number of layers can be used). However, this aspect is completely unrelated to the given paper.

7 Graphical language definition by means of metamodel specialization

In this section, we demonstrate how the proposed alternative metamodeling paradigm—the metamodel specialization can be used for the graphical syntax definition in an efficient way. We show that most of the desired syntax features can be defined relatively simply here, using only standard UML elements. The extension of the language definition to a complete editor definition also requires the same simple UML facilities.

As mentioned before, the starting point for the application of the specialization approach to a modeling task is the Universal Metamodel (UMM) for this task. Here we will define the Universal Metamodel for the graphical diagram syntax definition. The syntax of any concrete graphical language is defined by a specialization of this metamodel. Figure 15 shows the Universal Metamodel. In a sense it is similar to the TDA Type metamodel in Sect. 3—the choice of classes in this UMM has been to a great degree influenced by a positive experience in using TDA. However, diagram elements are used instead of their types and some more elements are added. The main semantic difference is that for a UMM specialization each permitted diagram element is a direct instance of a specialized class—a subkind of node, edge or compartment. This permits to apply all UML class model facilities—multiplicities, OCL constraints, etc., directly to the specialized metamodel. The new class CompartmentRow represents a line in a multiline compartment. The informal semantics of the compartment classes is similar to that in TDA—they represent any textual element or subelement in the concrete syntax of the language. The *value* attribute is added to compartment classes for writing OCL constraints on the textual value of a compartment or subcompartment. Thus, in fact, the specialization of this UMM for a language represents an alternative form of the context-free grammar of the textual syntax embedded in the graphics. Even more—the textual elements which are hidden (made invisible) in

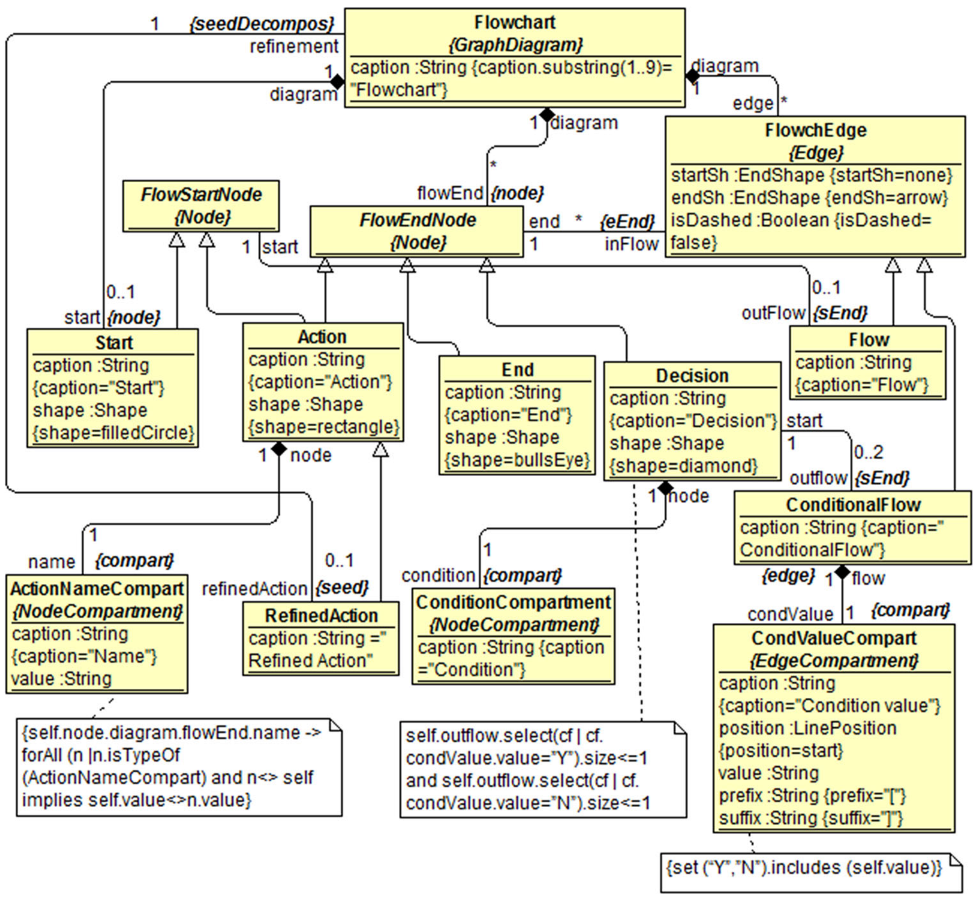

Flowchart
{GraphDiagram}
caption :String {caption.substring(1..9)="Flowchart"}
FlowchEdge
{Edge}
startSh :EndShape {startSh=none}
endSh :EndShape {endSh=arrow}
isDashed :Boolean {isDashed=false}
FlowStartNode
{Node}
FlowEndNode
{Node}
Start
caption :String {caption="Start"}
shape :Shape {shape=filledCircle}
Action
caption :String {caption="Action"}
shape :Shape {shape=rectangle}
End
caption :String {caption="End"}
shape :Shape {shape=bullsEye}
Decision
caption :String {caption="Decision"}
shape :Shape {shape=diamond}
Flow
caption :String {caption="Flow"}
ConditionalFlow
caption :String {caption="ConditionalFlow"}
ActionNameCompart
{NodeCompartment}
caption :String {caption="Name"}
value :String
RefinedAction
caption :String ="Refined Action"
ConditionCompartment
{NodeCompartment}
caption :String {caption ="Condition"}
CondValueCompart
{EdgeCompartment}
caption :String {caption="Condition value"}
position :LinePosition {position=start}
value :String
prefix :String {prefix="["}
suffix :String {suffix="]"}
{seedDecompos}
refinement
diagram
edge
flowEnd {node}
end {eEnd}
inFlow
start
outFlow {sEnd}
start {node}
node
outflow {sEnd}
name {compart}
refinedAction {seed}
condition {compart}
{edge}
flow
condValue {compart}
{self.node.diagram.flowEnd.name -> forAll (n |n.isTypeOf (ActionNameCompart) and n<> self implies self.value<>n.value}
self.outflow.select(cf | cf. condValue.value="Y").size<=1 and self.outflow.select(cf | cf. condValue.value="N").size<=1
{set ("Y","N").includes (self.value)}

Fig. 16 Flowchart syntax defined using metamodel specialization

this notation, but exist due to the fact that they are entered as independent entities by the user in the language editor context, should appear in the specialization as subcompartments. This is because we do not have the abstract syntax view of the language in our approach. The top class of the UMM— the GraphicalLanguage class, is used to represent the situation when the language to be defined contains several diagram types (e.g., as the UML does).

Now let us show the same Flowchart syntax definition using the metamodel specialization, see Fig. 16. This definition is shown in the custom notation introduced in the previous section, and this notation will be used for all subsequent specialization examples. The classes in the specialization represent graphical syntax elements of a Flowchart. The specialized classes retain only those inherited from UMM attributes which are really needed for the Flowchart definition. If an attribute must have a constant value, an OCL constraint is added to this attribute in the subclass. All required diagram element multiplicities are defined directly as multiplicities on the relevant redefined associations. In order to minimize the number of redefined associations in the specialization, two abstract node superclasses FlowStartNode and FlowEndNode are introduced in the specialization; in addition, one edge superclass FlowchEdge is used as well. These superclasses permit to specify the constraints expressing which edges can start from which node kinds in a compact way (in fact, a similar approach was already used in TDA using imitated subclasses in Sect. 3). The requirement that all action names in a diagram must be distinct is specified as an OCL constraint for the ActionNameCompart class. Another constraint specifies that no more than one conditional flow from a decision can be labeled by “Y” (and the same for ”N”). The third constraint specifies that no other strings may be used as labels. Here such constraints can be defined in a natural way, using only elements of the specialized metamodel—property names for navigation. Thus, all the desired Flowchart syntax features have been defined in the specialization in a direct and readable way. In order to define the action refinement feature, an Action subclass RefinedAction is introduced—the link *refinement* (redefining *seedDecomposition* in UMM) permits to reach the refining flowchart. While this feature may be not so important for flowcharts, the decomposition possibility included in UMM is vital for more complicated graphical languages. The flowchart example does not have a very complicated graphical syntax, and the whole language consists of just one diagram type (therefore, the GraphicalLanguage class is not specialized). But it has been checked that a complete graphical syntax definition of UML Class diagram notation can be defined this way—it takes about four pages in a readable resolution. The subcompartment concept permits to define also the structure of complicated text elements such as class attributes and operations with the same details as in the UML documentation. Some elements of this approach are demonstrated in Sect. 10 where graphical editor fragments for a UML class diagram subset are presented.

8 Graphical diagram editor definition using metamodel specialization

The graphical syntax definition approach can be easily extended to a graphical editor definition platform for the given diagram notation – the most natural executable behavior to be based on the diagram syntax. First, the Universal Metamodel has to be extended by some new classes and some new attributes have to be added to the existing classes—see Fig. 17. Attributes added for the editor definition are in bold font. Similarly, the new classes and two new enumerations have bold outlines in Fig. 17. However, the main new element is the concept of Universal Engine—a generic diagram editor whose generic behavior can be defined in terms of UMM, but the real behavior details depend on the given UMM specialization.

Typically, any real diagram editor contains the concept of project – a set of related diagrams having a common usage. Therefore, we also include project class in our UMM—it replaces the GraphicalLanguage class used in the UMM for syntax definition (we assume that the project can contain diagram types in a graphical language). 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, and the seed class for a diagram type will be a subclass of the Node class in the relevant specialization (therefore, no specific seed class in the UMM). To support the navigation from a seed node to a diagram the UMM association *seed—seedDecompos* is really used. However, this association may be used in other contexts as well, e.g., for action refinement in flowcharts already mentioned in Sect. 2. The new attributes are best to be explained when the UE is described.

The Universal Engine for diagram editors is an abstract editor whose generic behavior is explained in terms of UMM. But it is assumed that there exist one or more specializations of the UMM according to which UE really behaves. The behavior dynamics description certainly involves also the editor user whose actions actually determine the result. 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 the UE behavior will be defined just in terms of these actions. It is assumed that UE manages a project which contains diagrams of one or more types constituting a graphical language defined via the diagram syntax definition facilities described in Sect. 7. By means of the already mentioned project diagram containing seeds, the user can create a new diagram or open an existing diagram in order to modify it. When a new empty diagram of the given type is opened, a standard style Palette is opened as well. This Palette contains elements for all node types and edge types defined via Node and Edge subclasses present in the given specialization (more precisely, for all non-abstract subclasses). UE infers these elements from the specialization, and the concrete Palette element appearance (icon and textual identification) is specified via the new attributes of Node or Edge in UMM—*palIcon* and *palCaption*. It should be mentioned that the project diagram also has a palette for seeds of all diagram types defined in the specialization.

Abstract subclasses in the specialization are used just as “containers” for common attributes and associations for their concrete subclasses. When the user clicks on a palette element, UE creates the corresponding diagram element. 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. More style attributes such as background color, line color and line width are defined in the extended UMM in Fig. 17, and default values for these attributes can be defined in the specialization. UE then creates a node or edge with the given style, and it supports also an explicit modification of these “nonessential” style attributes by the user later on. In addition, UE checks the structure and multiplicity constraints in the specialization defined implicitly via the specialization as a UML class diagram or explicitly via OCL. If the user tries to violate a constraint, the new element is not created and a standard error message is displayed.

Another aspect supported by UE is the creation of textual elements in the diagram. For each node or edge to be created, UE opens a dialog form based on the specialized NodeCompartment or EdgeCompartment classes attached to the specialized diagram element class (if there are such). Similarly, this form is opened by UE when the user double-clicks a node or edge. The form contains elements for all compartments of the node or edge defined in the corresponding specialization. The main new attribute added in the UMM for this purpose is *inputContr*, which determines the input control type, which is offered to the user for entering the compartment value. Certainly, the supported types of input controls depend on the capabilities of the implemented 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

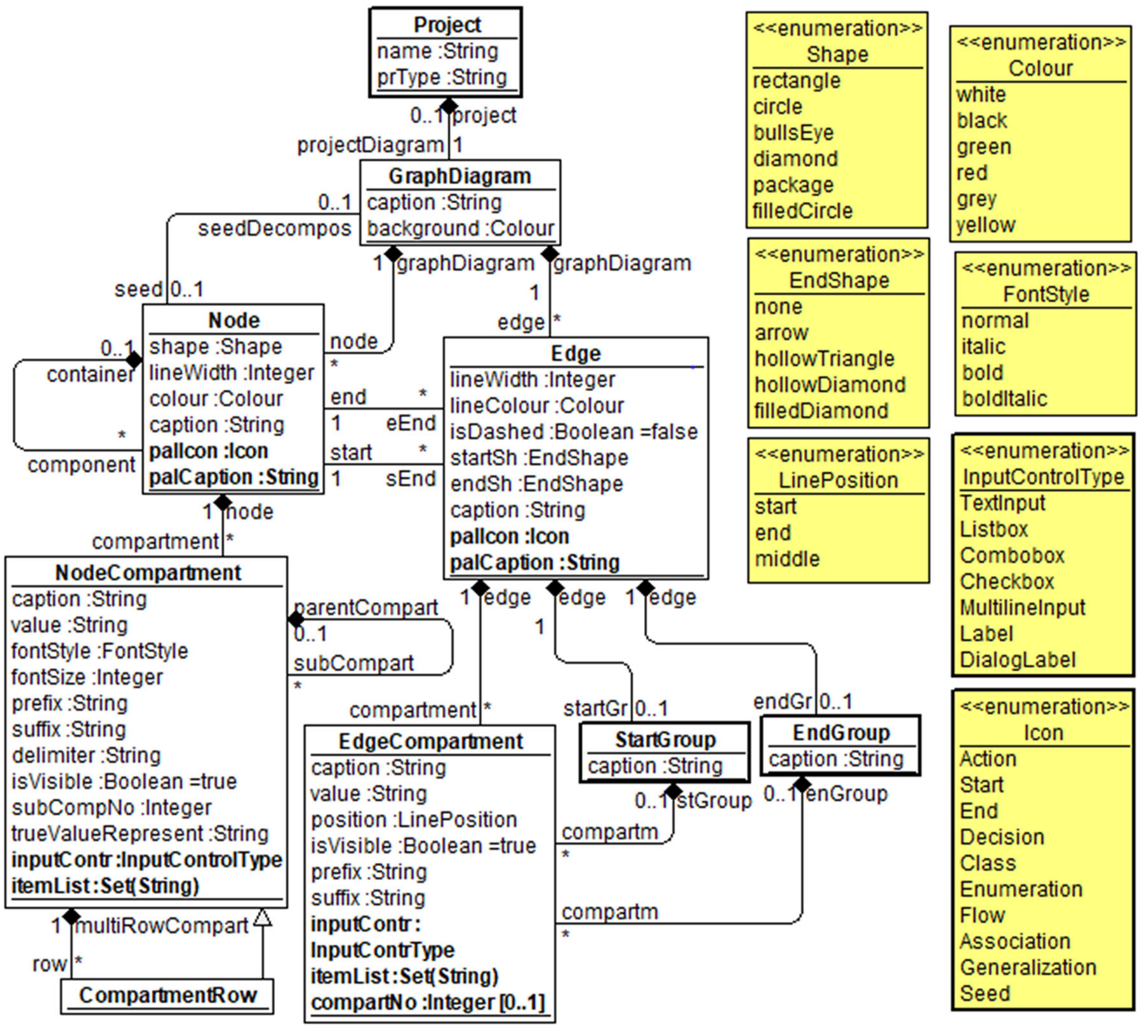


Fig. 17 UMM for graphical editor definition

these controls, there must be a possibility to define the appropriate value list; therefore, the *itemList* attribute of the type Set(String) is added (or of the type String [*] according to the strict UML syntax). The default value of this attribute must be set in the specialized compartment class (if listbox or combobox is selected for the compartment input), and this value may be a constant set or an OCL expression deriving the set from other diagram elements already created. Another non-trivial control type is MultiLineInput. This control is specially adjusted to creating node compartment texts consisting of logically independent lines (multiline compartments), such as attributes or operations in a Class node. There UE provides an independent entry of each line using the CompartmentRow class (a subclass of Compartment) in UMM which must be specialized for a multiline compartment. If the line has a more complicated structure than a simple text string, a new subform is opened for each new line entry. It should be reminded that the concept of multiline compartment and CompartmentRow was already present in the UMM for diagram syntax for defining such structures.

A compartment text may have a substructure, e.g., a class attribute text in UML consists of its name, type, default value, modifiers, etc. These elements are separated by constant prefixes or suffixes in the common string value, and 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. Therefore, UE for a compartment with subcompartments in the specialization creates a nested structure in the entry form. All these advanced UE features are demonstrated in Sect. 10 for an editor for a subset of class diagrams. In order to offer some structuring of the input form also for edge compartments, two compartment groups for compartments logically related to edge start or end are offered, see the UML association entry example in Sect. 10.

The whole input process of a compartment is organized by UE in a fixed way. The specialization may configure a compartment structure and the input control used for a compartment/subcompartment entry, and it may provide the required style values. In addition, a standard UML constraint (an OCL expression returning a Boolean value) may be added to the specialized compartment class to check the correctness of the entered compartment value after the user has completed the input of this compartment. We remind that this value is stored in the *value* attribute of the compartment.

We conclude the description of UE with some general behavior features. Upon start, 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 palette is shown. The user can add a new diagram of a supported kind by creating its seed from the palette or open an existing diagram—by double-clicking on the seed. Then, the editing of the diagram may start as described above. Besides this specialization-related UE behavior, UE offers some default behavior to the user—to save a project, to modify nonessential style attributes of a node or edge, to copy elements from diagram to diagram, to delete a diagram element, to modify the layout, etc.

Finally, to build UMM specializations described here, one more metamodel specialization facility must be permitted—the definition of a default value of an attribute. For graphical syntax definition, the only proposed way to specify attribute values was via OCL constraints on attributes. But for an editor definition it is very natural to define default values of attributes for specialized classes—according to UML semantics, these values are set when a new class instance (here—a diagram element) is created. For nonessential style attributes which can be modified by the user, this is the only correct way to define the values. But we allow also to use default values instead of constraints for other attributes as well, in order to obtain a more compact notation. The default value may be specified by a constant or OCL expression. In addition, some of the most typical default values can be set already in UMM (e.g., *isVisible* = *true* for compartments) and redefined in the specialization when needed.

9 Flowchart editor specialization example

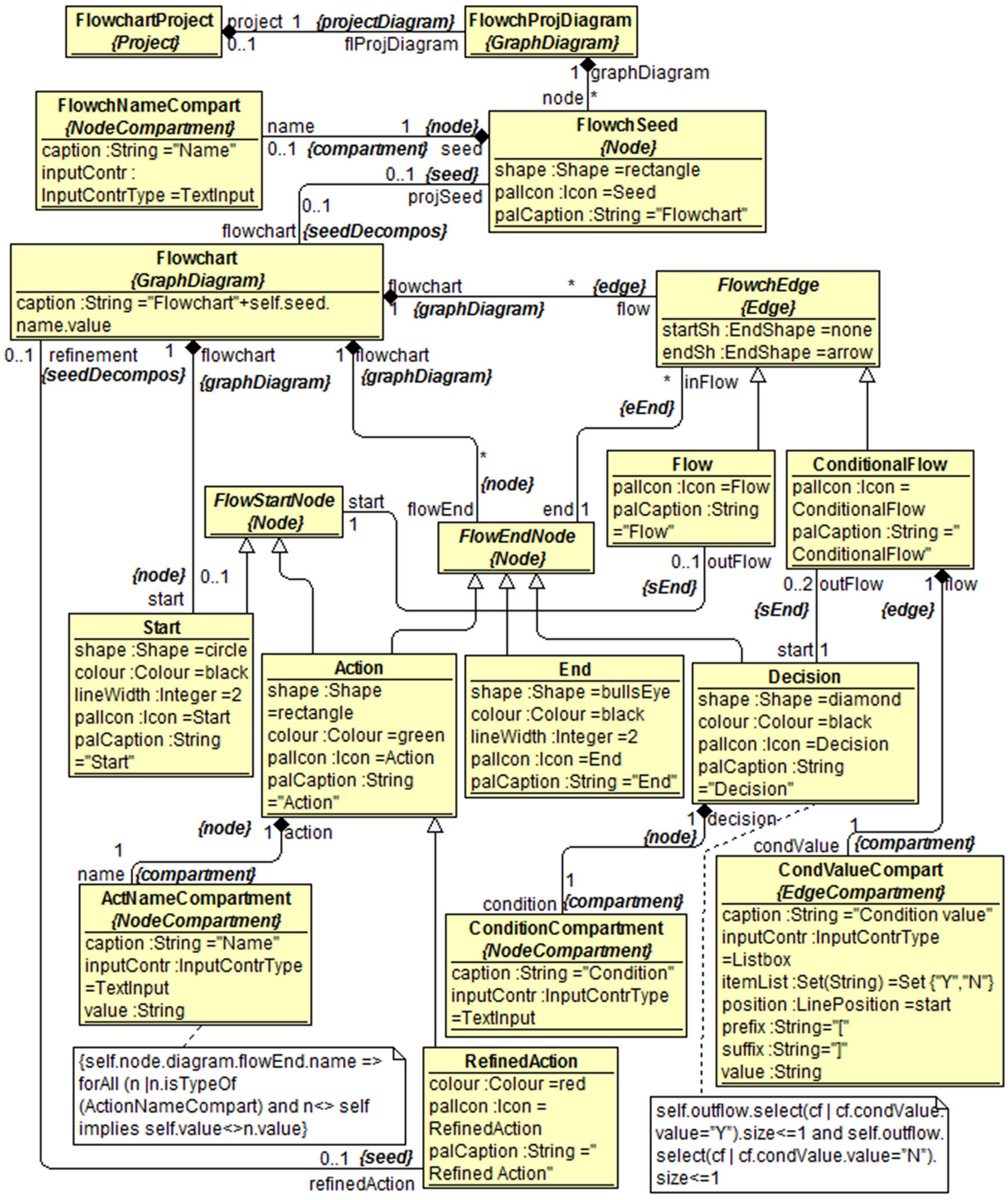
In this section, the editor definition facilities are demonstrated on a flowchart editor according to the flowchart diagram syntax defined in Sect. 7. For the editor UMM specialization examples, we will use here only the custom notation for the UMM class specialization introduced in Sect. 6. We recall that for association ends, where the original and redefined role names coincide, no explicit redefinition is required. Class inheritance within the specialization will be shown according to the standard UML notation.

Figure 18 shows the editor UMM specialization for the flowchart editor. There all required attribute settings are defined via the default value option, but not OCL constraints.

The project class from UMM is specialized to Flowchart-project with just one FlowchartProjectDiagram attached to it. This diagram contains named FlowchSeed nodes from which the corresponding Flowchart diagram instance can be opened. The created palette for the Flowchart project diagram contains only one element for the seed node. Thus, the user can create any number of flowcharts in the project. 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 with 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 Flowchart specialization for editor definition is quite similar to that for Flowchart syntax definition in Sect. 7, the same concrete and abstract classes are used. The generated palette for a Flowchart contains five node elements and two edge elements (for all non-abstract Node and Edge subclasses). The difference is that more attribute values have to be specified in this specialization. For all compartments, the *inputContr* attribute must be specified; for the condition value to be set on a conditional flow the use of listbox is demonstrated—the possible values are "Y" or "N," specified by a constant set expression. Since these values must be shown in square brackets in the diagram, the *prefix* and *suffix* attributes are set to the corresponding values. The *position* attribute specifies that the text must be positioned near to the edge start. The role of abstract superclasses in the specialization is the same as in the syntax definition—to reduce the number of redefined associations. The editor behavior on a new RefinedAction instance added to a flowchart is similar to that on a seed in the project diagram—upon double-click on it the refinement flowchart (with the given action name) is opened. UE enables this similarity of behavior because in both cases the same UMM association from a seed node to a diagram is redefined.

The OCL constraint attached to the Action name compartment now has a more active semantics—UE checks whether a new Action to be added to a Flowchart indeed has a name distinct from all existing action names in this Flowchart (if not, an error message is shown). Similarly, the editor checks that no more than one edge labeled by "Y" (or "N") exits from a decision.

The example shows that most of the editor features can be specified using standard UML class diagram facilities (including default attribute values) in the specialization, and explicit OCL constraints have to be used only for more complicated cases.



FlowchartProject {Project}
project 1 {projectDiagram}
0..1 flProjDiagram
FlowchProjDiagram {GraphDiagram}
1 graphDiagram
node *
FlowchNameCompart {NodeCompartment}
caption :String ="Name"
inputContr : InputContrType =TextInput
name 1 {node}
0..1 {compartment} seed
0..1 {seed}
projSeed
0..1
FlowchSeed {Node}
shape :Shape =rectangle
palIcon :Icon =Seed
palCaption :String ="Flowchart"
flowchart {seedDecompos}
Flowchart {GraphDiagram}
caption :String ="Flowchart"+self.seed.name.value
flowchart * {edge}
1 {graphDiagram} flow
FlowchEdge {Edge}
startSh :EndShape =none
endSh :EndShape =arrow
0..1 refinement {seedDecompos}
1 flowchart {graphDiagram}
1 flowchart {graphDiagram}
* inFlow {eEnd}
* {node}
Flow
palIcon :Icon =Flow
palCaption :String ="Flow"
ConditionalFlow
palIcon :Icon = ConditionalFlow
palCaption :String =" ConditionalFlow"
FlowStartNode {Node}
start 1
flowEnd
end 1
FlowEndNode {Node}
0..1 outFlow {sEnd}
0..2 outFlow {sEnd}
1 flow {edge}
{node} 0..1 start
Start
shape :Shape =circle
colour :Colour =black
lineWidth :Integer =2
palIcon :Icon =Start
palCaption :String ="Start"
Action
shape :Shape =rectangle
colour :Colour =green
palIcon :Icon =Action
palCaption :String ="Action"
End
shape :Shape =bullsEye
colour :Colour =black
lineWidth :Integer =2
palIcon :Icon =End
palCaption :String ="End"
start 1
Decision
shape :Shape =diamond
colour :Colour =black
palIcon :Icon =Decision
palCaption :String ="Decision"
{node} 1 action
1 name {compartment}
1 decision {node}
condValue 1 {compartment}
1 condition {compartment}
ActNameCompartment {NodeCompartment}
caption :String ="Name"
inputContr :InputContrType =TextInput
value :String
ConditionCompartment {NodeCompartment}
caption :String ="Condition"
inputContr :InputContrType =TextInput
CondValueCompart {EdgeCompartment}
caption :String ="Condition value"
inputContr :InputContrType =Listbox
itemList :Set(String) =Set {"Y","N"}
position :LinePosition =start
prefix :String="["
suffix :String="]"
value :String
{self.node.diagram.flowEnd.name => forAll (n |n.isTypeOf (ActionNameCompart) and n<> self implies self.value<>n.value}
RefinedAction
colour :Colour =red
palIcon :Icon = RefinedAction
palCaption :String =" Refined Action"
0..1 {seed}
refinedAction
self.outflow.select(cf | cf.condValue.value="Y").size<=1 and self.outflow.select(cf | cf.condValue.value="N").size<=1

Fig. 18 Definition of flowchart graphical editor

10 Fragments of a simplified class diagram editor example

Now let us consider a more complicated and also a more realistic example. This section presents a basic fragment of a simplified UML class diagram editor. This editor supports the subset of UML which corresponds to class diagram features included in the EMOF metamodel [1] for the abstract syntax. In the class diagram, according to EMOF specification, Class and Enumeration nodes and Association and Generalization edges are supported. The Class node in turn includes all features for the EMOF support. Figure 19 shows the editor features related to attributes in a ClassNode—the multiline Attribute compartment and all details of an Attribute (name, type, multiplicity, default value, IsDerived feature and basic modifiers—readOnly, ordered and isIdent). In addition, the Class name and isAbstract compartments are also included—to illustrate the specific notation for showing that a class is

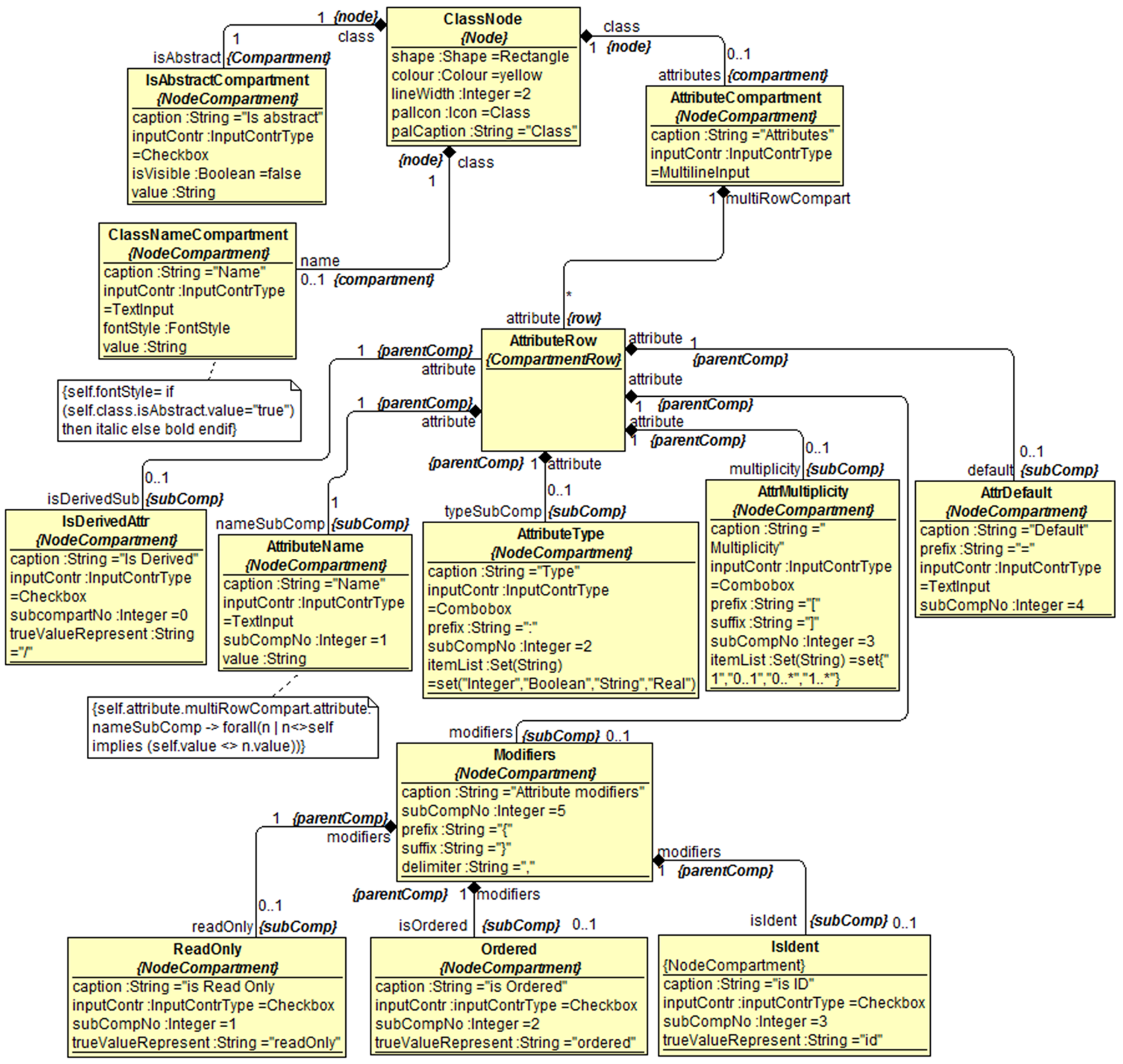

ClassNode {Node}
shape :Shape =Rectangle
colour :Colour =yellow
lineWidth :Integer =2
palIcon :Icon =Class
palCaption :String ="Class"
IsAbstractCompartment {NodeCompartment}
caption :String ="Is abstract"
inputContr :InputContrType =Checkbox
isVisible :Boolean =false
value :String
AttributeCompartment {NodeCompartment}
caption :String ="Attributes"
inputContr :InputContrType =MultilineInput
ClassNameCompartment {NodeCompartment}
caption :String ="Name"
inputContr :InputContrType =TextInput
fontStyle :FontStyle
value :String
{self.fontStyle= if (self.class.isAbstract.value="true") then italic else bold endif}
AttributeRow {CompartmentRow}
IsDerivedAttr {NodeCompartment}
caption :String ="Is Derived"
inputContr :InputContrType =Checkbox
subcompartNo :Integer =0
trueValueRepresent :String ="/"
AttributeName {NodeCompartment}
caption :String ="Name"
inputContr :InputContrType =TextInput
subCompNo :Integer =1
value :String
AttributeType {NodeCompartment}
caption :String ="Type"
inputContr :InputContrType =Combobox
prefix :String =":"
subCompNo :Integer =2
itemList :Set(String) =set("Integer","Boolean","String","Real")
AttrMultiplicity {NodeCompartment}
caption :String ="Multiplicity"
inputContr :InputContrType =Combobox
prefix :String ="["
suffix :String ="]"
subCompNo :Integer =3
itemList :Set(String) =set{"1","0..1","0..*","1..*"}
AttrDefault {NodeCompartment}
caption :String ="Default"
prefix :String ="="
inputContr :InputContrType =TextInput
subCompNo :Integer =4
{self.attribute.multiRowCompart.attribute.nameSubComp -> forall(n | n<>self implies (self.value <> n.value))}
Modifiers {NodeCompartment}
caption :String ="Attribute modifiers"
subCompNo :Integer =5
prefix :String ="{"
suffix :String ="}"
delimiter :String =","
ReadOnly {NodeCompartment}
caption :String ="is Read Only
inputContr :InputContrType =Checkbox
subCompNo :Integer =1
trueValueRepresent :String ="readOnly"
Ordered {NodeCompartment}
caption :String ="is Ordered"
inputContr :inputContrType =Checkbox
subCompNo :Integer =2
trueValueRepresent :String ="ordered"
IsIdent {NodeCompartment}
caption :String ="is ID"
inputContr :inputContrType =Checkbox
subCompNo :Integer =3
trueValueRepresent :String ="id"
class 1 {node}
isAbstract 1 {Compartment}
attributes 0..1 {compartment}
1 multiRowCompart
name 0..1 {compartment}
attribute * {row}
1 {parentComp} attribute
0..1 isDerivedSub {subComp}
1 nameSubComp {subComp}
0..1 typeSubComp {subComp}
0..1 multiplicity {subComp}
0..1 default {subComp}
modifiers {subComp} 0..1
readOnly {subComp} 0..1
isOrdered {subComp} 0..1
isIdent {subComp} 0..1

Fig. 19 The attribute fragment of EMOF class editor definition

abstract. Figure 20 shows the basic editor features related to AssociationEdge (association name and role, multiplicity and aggregation for both ends). The complete definition of a class diagram editor for EMOF subset would require two more diagrams of approximately the same size (Class Operations and Class generalization, Instances + Enumerations).

The main goal of the example in Fig. 19 is to illustrate how the given UMM specialization approach can be used for the relatively complicated structure of an Attribute in a class diagram. All already mentioned subcompartment features are used here. In fact, some more features are necessary—they were already included in the UMM in Fig. 17, but not explained so far. The complete definition of Attribute syntax requires two-level subcompartments – there is a list of Modifiers of several kinds in braces as part of an Attribute row. To support such lists of options, compartment in UMM includes also the delimiter attribute (for Modifiers the delimiter is the "," character). Actually, two-level subcompartments and delimiters are needed for other class diagram elements as well, e.g., for Operation parameters. One more feature already present in our UMM is the support for visual presentation of optional parts in the final compartment value—the *trueValueRepresent* attribute for NodeCompartment class. In the editor dialog form, these parts typically are represented

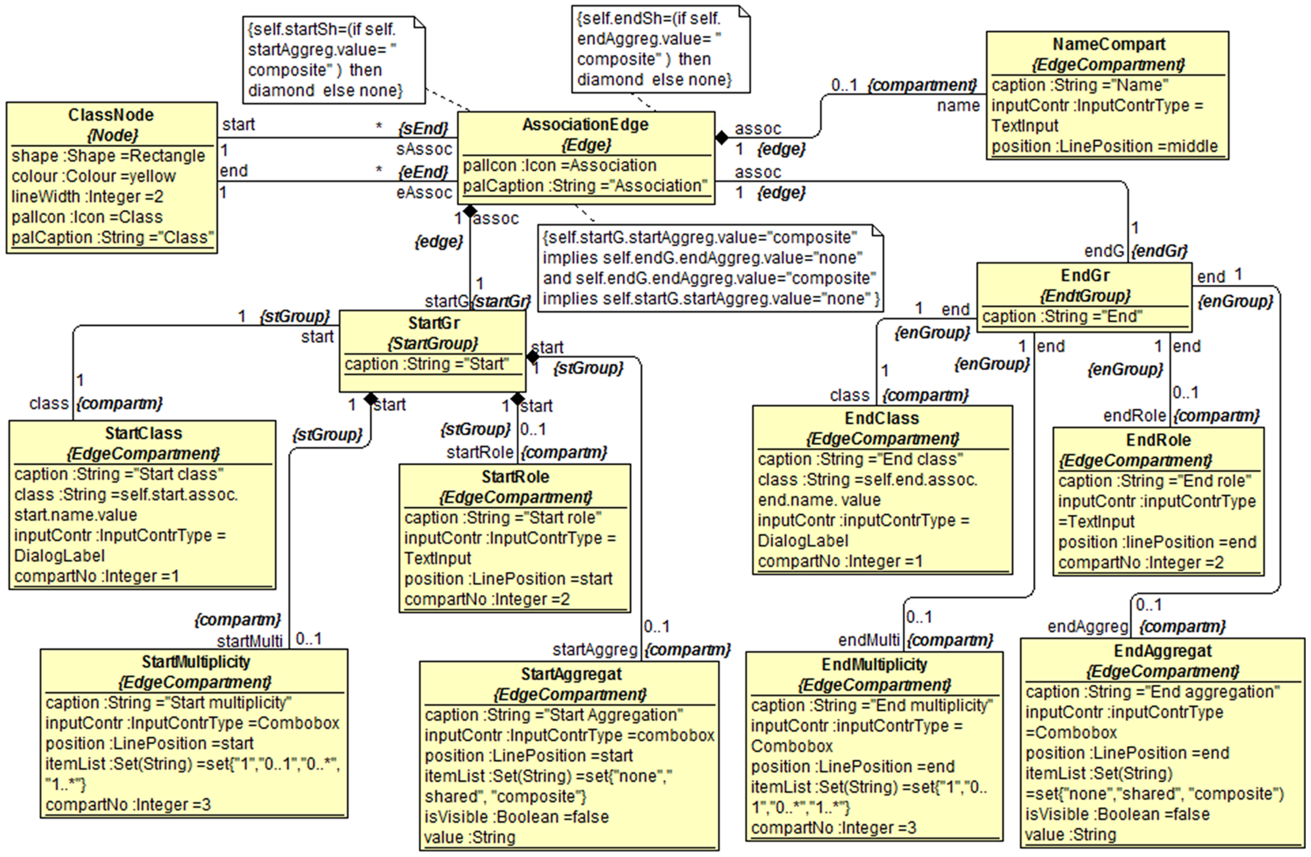

{self.startSh=(if self. startAggreg.value=" composite") then diamond else none}
{self.endSh=(if self. endAggreg.value=" composite") then diamond else none}
ClassNode {Node} shape :Shape =Rectangle colour :Colour =yellow lineWidth :Integer =2 palIcon :Icon =Class palCaption :String ="Class"
AssociationEdge {Edge} palIcon :Icon =Association palCaption :String ="Association"
NameCompart {EdgeCompartment} caption :String ="Name" inputContr :InputContrType = TextInput position :LinePosition =middle
{self.startG.startAggreg.value="composite" implies self.endG.endAggreg.value="none" and self.endG.endAggreg.value="composite" implies self.startG.startAggreg.value="none" }
StartGr {StartGroup} caption :String ="Start"
EndGr {EndtGroup} caption :String ="End"
StartClass {EdgeCompartment} caption :String ="Start class" class :String =self.start.assoc. start.name.value inputContr :InputContrType = DialogLabel compartNo :Integer =1
StartRole {EdgeCompartment} caption :String ="Start role" inputContr :InputContrType = TextInput position :LinePosition =start compartNo :Integer =2
EndClass {EdgeCompartment} caption :String ="End class" class :String =self.end.assoc. end.name. value inputContr :inputContrType = DialogLabel compartNo :Integer =1
EndRole {EdgeCompartment} caption :String ="End role" inputContr :inputContrType =TextInput position :linePosition =end compartNo :Integer =2
StartMultiplicity {EdgeCompartment} caption :String ="Start multiplicity" inputContr :InputContrType =Combobox position :LinePosition =start itemList :Set(String) =set{"1","0..1","0..*", "1..*"} compartNo :Integer =3
StartAggregat {EdgeCompartment} caption :String ="Start Aggregation" inputContr :InputContrType =combobox position :LinePosition =start itemList :Set(String) =set{"none"," shared", "composite"} isVisible :Boolean =false value :String
EndMultiplicity {EdgeCompartment} caption :String ="End multiplicity" inputContr :inputContrType = Combobox position :LinePosition =end itemList :Set(String) =set{"1","0.. 1","0..*","1..*"} compartNo :Integer =3
EndAggregat {EdgeCompartment} caption :String ="End aggregation" inputContr :inputContrType =Combobox position :LinePosition =end itemList :Set(String) =set{"none","shared", "composite"} isVisible :Boolean =false value :String
start 1 * {sEnd} sAssoc
end 1 * {eEnd} eAssoc
assoc 1 {edge}
assoc 1 {edge}
0..1 {compartment} name
1 assoc {edge}
1 startG {startGr}
1 endG {endGr}
1 {stGroup} start
1 class {compartm}
1 start {stGroup}
1 start {stGroup} 0..1 startRole {compartm}
start 1 {stGroup}
{compartm} startMulti 0..1
0..1 startAggreg {compartm}
1 end {enGroup}
end 1 {enGroup}
1 class {compartm}
1 end {enGroup}
1 end {enGroup} 0..1 endRole {compartm}
0..1 endMulti {compartm}
0..1 endAggreg {compartm}

Fig. 20 The Association fragment of EMOF class editor definition

by a checkbox control, but, if the user selects true, typically some string must be included in the complete value, e.g., "ordered" is specified for the *Ordered* subcompartment.

A mixed notation for UMM attribute specialization is used in Fig. 19. Attributes 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 (font style for Class name). In addition, one entered value check is shown as an OCL constraint—the uniqueness of attribute names per class. The constraints typically are based on the *value* attribute for a compartment class. Though this attribute is in fact present for all such classes in the specialization (it is inherited from UMM), in examples we show this attribute only for those classes where it is referenced in OCL constraints.

Now some more detailed comments on interesting features in the editor definition are given. 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 –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 IsAbstractCompartment value). The presence of all textual elements of the language syntax in the specialization permits to define rich OCL constraints, which can span contents of several related diagram elements, without the need to use an independent abstract syntax definition.

The multiline Attribute compartment is to be created by the user via the MultiLineInput control. Therefore, UE provides an independent entry of each line using the AttributeRow class which is a subclass of CompartmentRow in UMM. On the one hand, UE for this feature provides a multiline text view of the whole multiline compartment, with the possibility to add or remove lines. But since the CompartmentRow (and consequently also the specialized AttributeRow) is a subclass of Compartment in UMM, UE supports also all the functionality for Compartment here. In particular, a row may have subcompartments—in the given case there are six subcompartments – for entering IsDerived, attribute name, attribute type, multiplicity, default value and modifiers (the order of subcompartments is defined via the subCompNo attribute). UE for each line can show a dialog subform containing controls for subcompartment input—either for initial entry or value modification. To obtain the final value of a line, the subcompartment values are concatenated in the given order using the defined prefix or suffix, e.g., the ":" character is inserted before the attribute type, as required by the UML syntax. A prefix or suffix is not inserted if the value has not been entered by the user (this is possible for subcompartments with the multiplicity 0..1). The Attribute-Type compartment is to be entered via a combobox, with the item list showing the most typical values—UML primitive types, but any other type value can be entered as well. The Modifiers subcompartment involves the second level subcompartments—three options, which all may be present or not (entered via checkbox). For each option, the visible string in the case of presence is shown, and the options are separated by "," in the final resulting string.

The only proper value check on input by the user here is the uniqueness of attribute names, see the attached OCL constraint. The constraint is evaluated by UE when the user has completed the value entry (moved away from the control), at both initial value input and modification.

Now some comments on the Association fragment of the editor in Fig. 20. The Class node definition is repeated in the fragment to represent the association ends in a class diagram, and UE automatically selects the appropriate instances according to user selection in the diagram. The name compartment is attached directly to the Association edge, but other compartments—role, multiplicity and aggregation for both ends, are attached to the start and end groups, respectively. The groups in the association form are shown via tabs. In order to remind the user to which class in the diagram the respective association end is attached, the groups start with a read-only compartment displaying the class name. Two of the OCL constraints control how the association ends must be visualized, but the third one prohibits wrong aggregation settings for both ends by the user (since it is attached to the AssociationEdge class, it is checked when the user has completed the association editing).

The provided examples confirm the fact that 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. However, the approach is mainly oriented toward such graphical DSL support where typically only features similar to those shown here are required.

11 Extension of the editor for declarative mapping to abstract syntax

In this section, we demonstrate an additional task to be naturally solved by metamodel specialization. But first some general comments on tasks for which metamodel specialization is well applicable. In fact, these are families of similar model processing tasks where each specific task can be defined solely by a specialization of the UMM for the family. The UMM for the family describes common semantics for the tasks in the family. The corresponding UE implements dynamic aspects of this semantics—it should be defined to work on instances of UMM, and the specialized behavior should be dependent only on (fixed) values of some attributes of UMM classes. It should be noted that not all model-related tasks are in this category; for example, tasks to be solved by "classic" model transformations frequently are best to be defined on non-specialized metamodels.

To take all this into account, the task of declarative mapping of the graphical syntax of a language to its abstract syntax is very adequate since the graphical syntax is already defined by metamodel specialization. And it is natural to attach this task to the graphical editor definition in order to obtain a synchronous building of the abstract syntax model according to a classical EMOF metamodel. In many existing graphical editor frameworks, such as Eclipse GMF [4], all the functionality is based on the abstract syntax of the language to be supported and on mapping this syntax to the graphical one. On the contrary, our approach to graphical languages is based directly on the graphical syntax. But there are tasks where the parallel abstract syntax representation is required. If the language has an executable semantics, this semantics as a rule is based on the abstract syntax. Therefore, compiler or interpreter support for the language frequently requires this syntax as well. Even the OMG Diagram Definition (DD) and Diagram Interchange (DI) standard [3] requires a mapping to abstract syntax to be maintained, therefore abstract syntax is required even for ensuring correct interchange of diagrams created in our editors with other classic tools.

Frequently, a mapping between graphical and abstract syntax of a language is maintained by some executable code, such as model transformations. But here we offer a completely declarative definition of such mapping based on the specialization of an extended UMM including also the mapping part (Fig. 21). This UMM in fact is a direct extension of UMM for editors in Fig. 17. To make things simpler here, we assume that the mapping from graphical syntax to abstract syntax is one-to-one. For each main graphical syntax class (Node or Edge) instance, we assign at first an instance of the corresponding mapping class which, in turn, is mapped to an instance of the corresponding abstract syntax class. This is completely in line with a simple editor behavior – as soon as a new graphical element is created, its counterpart in abstract syntax is created as well. Certainly, in some situations things are a bit more complicated (especially for compartments), and this will be explained later.

In our approach (based on metamodel specialization), a specific situation is how to include in the mapping definition the relevant part of an existing metamodel for abstract

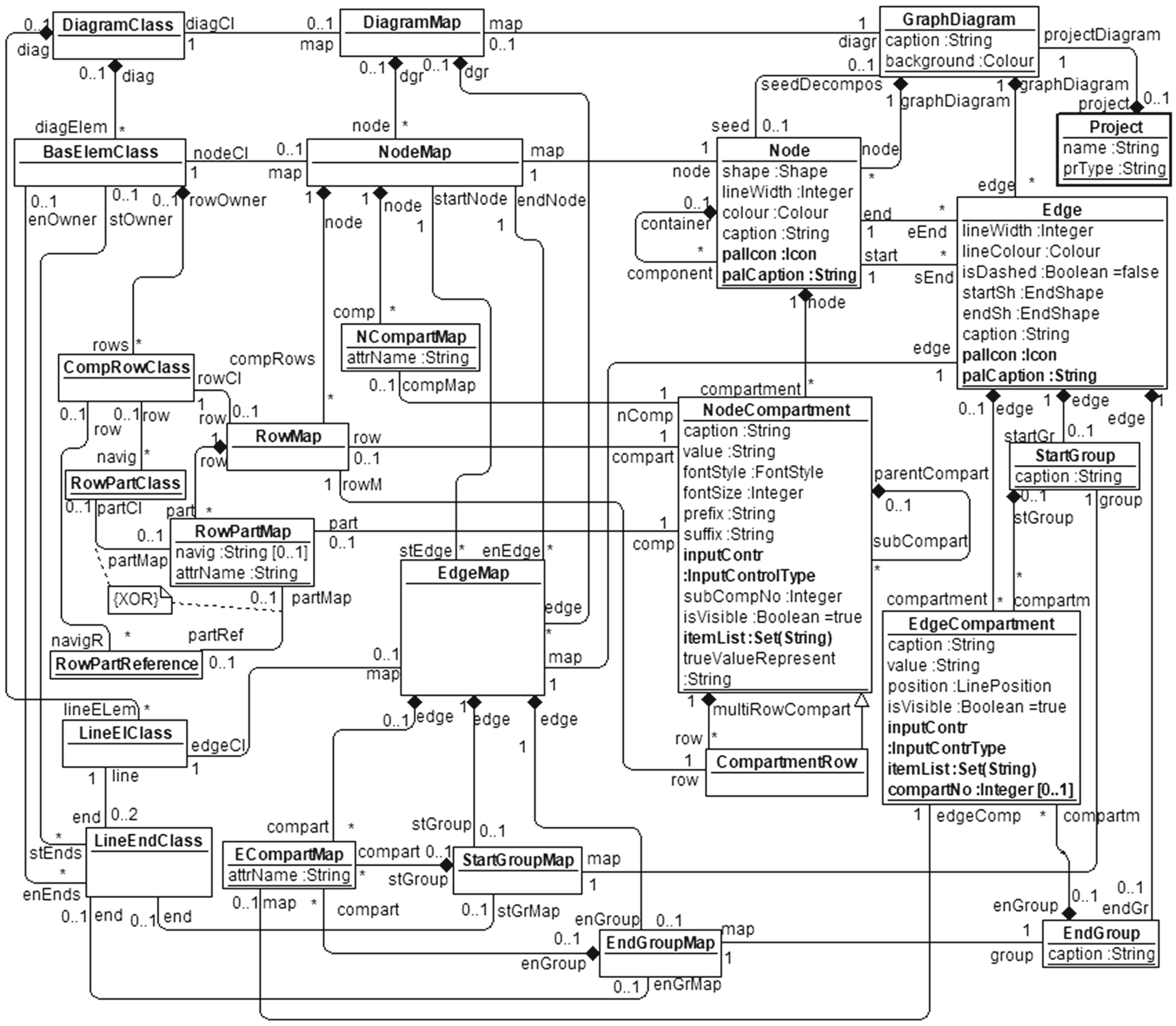


Fig. 21 Extended UMM for editor and mappings

syntax. For example, this is required for defining such mapping from our class diagram fragment (EMOF level) to the standard UML metamodel for that part of class diagrams. Figure 22 shows a very minimal fragment of the UML abstract syntax metamodel relevant for class diagram features used in our examples (taken from the EMOF documentation [1]). The fragment contains a number of abstract classes whose only goal is to lift common attributes or associations as high as possible. The fragment shows how the *name* attribute is inherited from NamedElement to classes really used in our mapping examples (Class, Property, Association). This situation is very typical to UML metamodel, which makes this metamodel quite difficult to understand. In fact, some other attributes of Property are defined in abstract superclasses as well, but we simplify the fragment in order to minimize the example size, since the situation is very similar to *name*.

Since the reuse of existing metamodel fragments in a specialization is different from the direct "in-place" specialization of UMM classes, a slight extension of our metamodel specialization features is required for this task. This extension will be explained in detail when discussing the extended UMM and its use for mapping examples. However, the proper mapping "infrastructure" part can be defined by UMM specialization features already described. We define a mapping for a graphical language as a set of classes and associations, providing at instance level for each element of the graphical syntax a link to a mapping instance from which another link leads to an instance in the abstract syntax model. At first, we show the extended UMM for editors including also the mapping support. Figure 21 presents this extended UMM. The right part of it contains the main classes of the UMM for editors (from Fig. 17). The left part of the UMM represents the proper mapping metamodel (classes with the suffix Map). Classes on the far left (with the suffix Class) are a sort of metamodel class templates to be used as formal superclasses of the existing abstract syntax metamodel (ASMM) classes. Such template classes in UMM have no attributes at all. Their role is to attach the required associations from mapping classes to the relevant ASMM classes in the specialization (in a way usable by UE). To attach an association to an ASMM class, we simply have to specify in the specialization that this ASMM class is a subclass of the corresponding template class in UMM. Then at runtime, the extended UE for each new graphical element instance in editor will create also the corresponding linked mapping instance, a new ASMM instance template (with no attributes at all) and the link from the mapping instance to this template.

Another new specialization feature used in this UMM is the possibility to reference an attribute (of an ASMM class) whose name is not defined in the UMM but appears only in the specialization. In fact, the mapping classes in UMM are of two kinds—main ones (NodeMap, RowMap, EdgeMap, StartGroupMap, EndGroupMap) that have a direct link to an ASMM class and compartment related (NCompartMap, RowPartMap, ECompartMap) for which such link is not present (or optional). These compartment-related classes have an additional attribute *attrName*, which in the specialization points to a specific attribute of the ASMM class to which the parent main mapping class is linked. The informal dynamic semantics of this construct is that UE will add an attribute with the specified name (and the value obtained via the mapping instance from the compartment value) to the ASMM instance template created for the compartment parent map. However, formally this action is performed in a more complicated way since UE cannot directly add an attribute not defined in the UMM to an UMM class instance—see the technical details in Sect. 12. For the RowPartMap class, the situation is even more complicated. First, it can be mapped to an attribute of the parent mapping (RowMap) target class in the ASMM. But it can be mapped also to an attribute of a separate ASMM class (only linked to the parent mapping target). Therefore, this class has one more optional *navig* attribute which in the specialization can be set to the role name by which to find the required ASMM class (from the ASMM class linked to the parent). However, there are two distinct subcases—either the additional reference class instance already exists in the AS model and must be found using the value of the specified attribute (via *attrName*) or this class instance must be created when the mapping instance is created by UE. Therefore, two distinct ASMM template classes are included in the UMM—RowPartReference and RowPartClass. In a specialization, one of them must be

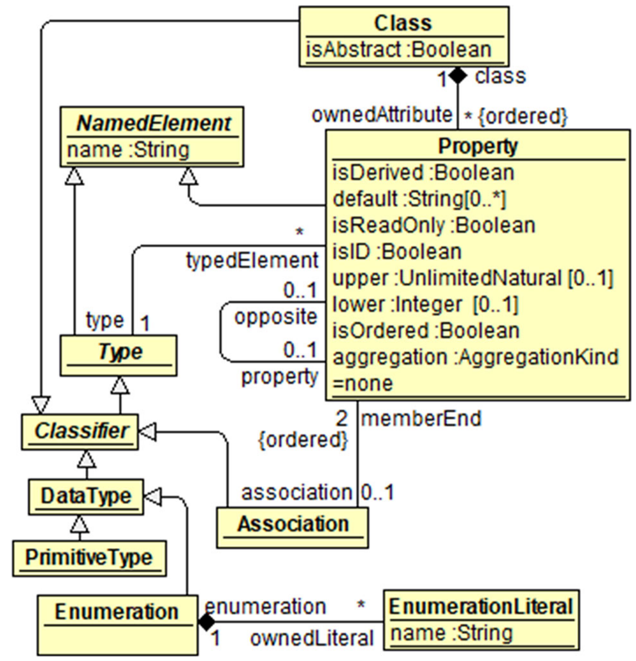


Fig. 22 Fragment of UML metamodel used in mapping examples

chosen—certainly the RowPartReference class for the first subcase. But in both cases the class instance must be attached by UE via the specified link. Further details of these new specialization features will be explained on examples.

Two example fragments will be given showing how Class attribute and association fragments can be mapped to the fragment of UML metamodel in Fig. 22. Though all the presentation in this section is related to class editor example, the proposed UMM and the related UE is in no way specific to this example. It has been checked that an editor and a mapping for UML activity diagrams can also be defined using this approach (in fact, for a subset of UML activity notation, since the complete activity notation is no more a pure graph diagram). Similar experiments were made for some workflow modeling notations.

We start with the mapping specialization example for the EMOF class editor attribute fragment in Fig. 19. Figure 23 shows how this editor definition can be extended to a related mapping definition. For each specialized diagram element in the class diagram editor fragment, the corresponding specialized mapping class is presented. In order to reduce the figure size, we do not repeat here all class attribute features from Fig. 19, but only those which require some unique facilities to define their mappings (others are similar).

The far left part of this specialization contains the small fragment of the UML metamodel already shown in Fig. 22 in the role of ASMM. Certainly, main classes here are Class and Property. The Class metaclass plays the role of BaseElemClass in the UMM, and the ClassMap class in the specialization is directly linked to it—via the *class* link

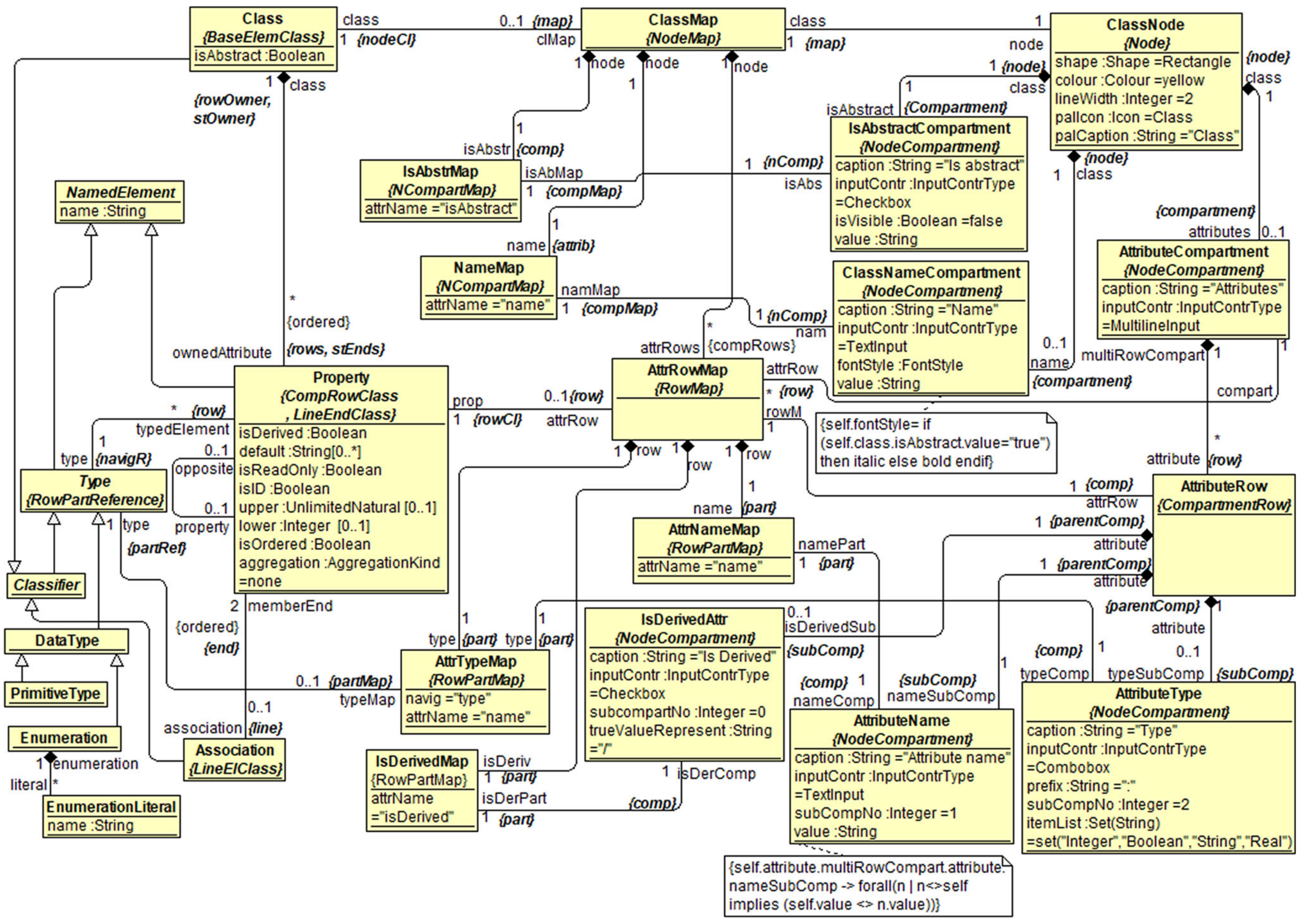


Fig. 23 Mapping specialization for the attribute fragment of EMOF class editor

which redefines the *nodeCl* link in UMM. Thus, the natural semantics of a Class node in a class diagram is enabled—it corresponds to a Class in the UML metamodel.

We continue with the details of a class node. Two "simple" Class node compartments—*isAbstract* and *name*, are shown with their mappings, and they are mapped to the UML Class attributes *isAbstract* and *name*, respectively (using the specialized values of *attrName*). Further, the multiRow Attribute compartment together with its Attribute row is mapped using the AttrRowMap class (this class is linked to both the compartment and row classes). The attribute row is mapped to the ASMM class Property – the natural counterpart of class attribute in the UML abstract syntax. To do this, Property is made a subclass of the UMM class CompRowClass. For the row, the mapping of three parts is shown—the attribute *name*, *type* and *isDerived*. The *name* and *isDerived* are mapped to the corresponding attributes of Property. But the *type* is mapped to the name of the ASMM class Type—this class is made a subclass of RowPartReference. The reference case must be used here since an attribute can have only a type existing in the model. Certainly, the Type class is an abstract superclass, so a concrete subclass of it must be found having the required name—most frequently either a PrimitiveType or an Enumeration instance. This search is done automatically by UE (in fact its component SE operating on the specialized model) when the mapping instance is created and the type instance is linked to Property via the specified *navig* link *type*. If such an instance is not found, an error message is shown. We want to emphasize once more that the compartment structuring facilities in our UMM are sufficient to define in the specialization any required mapping of text parts to attributes in ASMM.

One more issue here is the compartment value type compatibility with the type of the corresponding ASMM class attribute. The compartment value type in UE always is String, but the attribute type may be any type available in EMOF. In all normal situations, the value correspondence rule can be described by an OCL constraint in the specialization. However, in many typical situations the specialization definer's task can be simplified—UE can infer automatically the required value transformation from the input control used for the compartment value entry in the editor and the attribute type (they should be in a way compatible). No transformation is required if the attribute also has the String type. Similarly, if

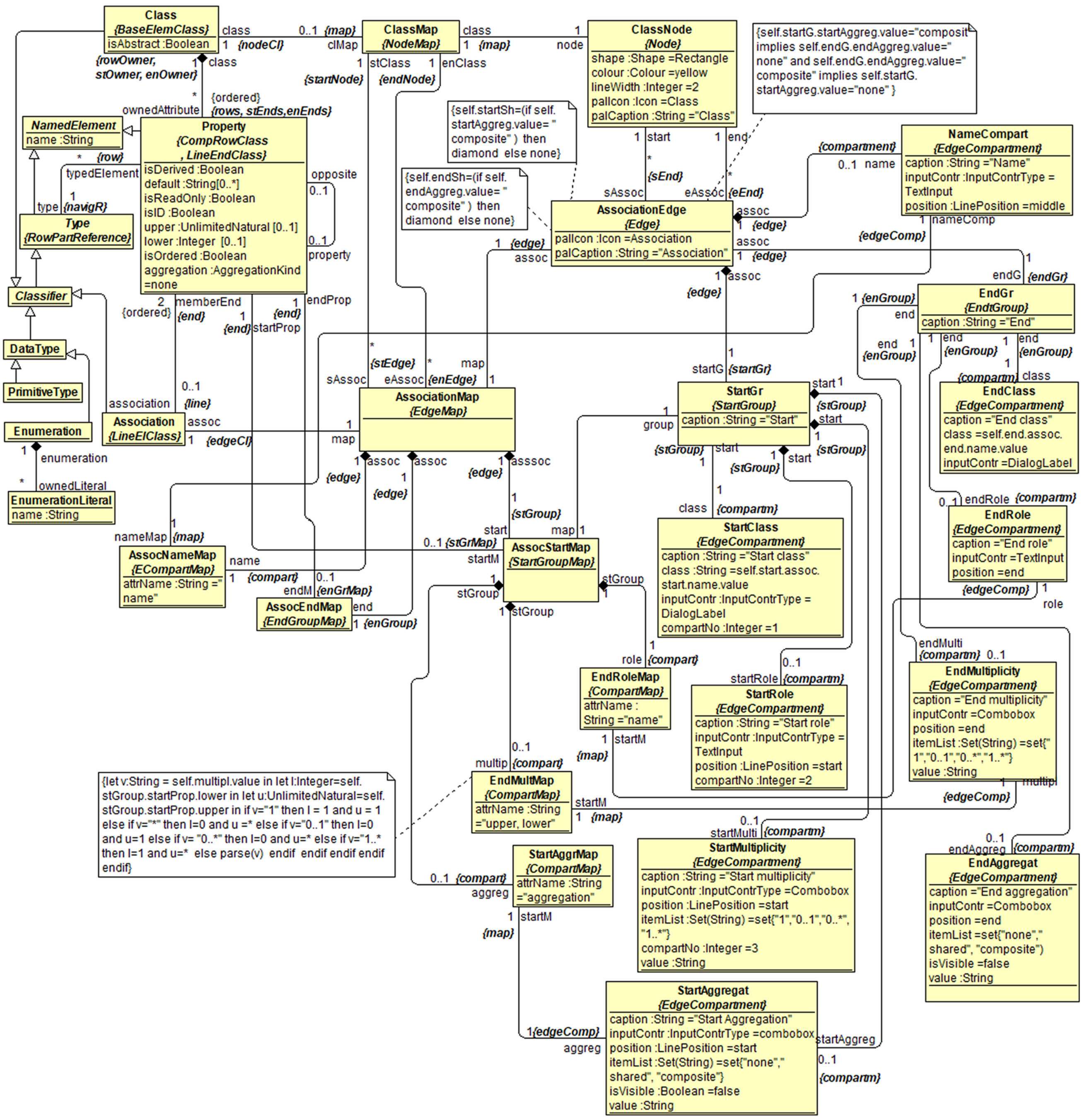

Class {BaseElemClass} isAbstract :Boolean
ClassMap {NodeMap}
ClassNode {Node} shape :Shape =Rectangle colour :Colour =yellow lineWidth :Integer =2 palIcon :Icon =Class palCaption :String ="Class"
{self.startG.startAggreg.value="composit" implies self.endG.endAggreg.value="none" and self.endG.endAggreg.value="composite" implies self.startG.startAggreg.value="none" }
NamedElement name :String
Property {CompRowClass , LineEndClass} isDerived :Boolean default :String[0..*] isReadOnly :Boolean isID :Boolean upper :UnlimitedNatural [0..1] lower :Integer [0..1] isOrdered :Boolean aggregation :AggregationKind =none
Type {RowPartReference}
Classifier
DataType
PrimitiveType
Enumeration
EnumerationLiteral name :String
Association {LineElClass}
{self.startSh=(if self.startAggreg.value="composite") then diamond else none}
{self.endSh=(if self.endAggreg.value="composite") then diamond else none}
AssociationEdge {Edge} palIcon :Icon =Association palCaption :String ="Association"
NameCompart {EdgeCompartment} caption :String ="Name" inputContr :InputContrType = TextInput position :LinePosition =middle
EndGr {EndGroup} caption :String ="End"
EndClass {EdgeCompartment} caption ="End class" class =self.end.assoc.end.name.value inputContr =DialogLabel
AssociationMap {EdgeMap}
StartGr {StartGroup} caption :String ="Start"
StartClass {EdgeCompartment} caption :String ="Start class" class :String =self.start.assoc.start.name.value inputContr :InputContrType = DialogLabel compartNo :Integer =1
EndRole {EdgeCompartment} caption ="End role" inputContr =TextInput position =end
AssocNameMap {ECompartMap} attrName :String ="name"
AssocEndMap {EndGroupMap}
AssocStartMap {StartGroupMap}
EndRoleMap {CompartMap} attrName : String ="name"
StartRole {EdgeCompartment} caption :String ="Start role" inputContr :InputContrType = TextInput position :LinePosition =start compartNo :Integer =2
EndMultiplicity {EdgeCompartment} caption ="End multiplicity" inputContr =Combobox position =end itemList :Set(String) =set{"1","0..1","0..*","1..*"} value :String
{let v:String = self.multipl.value in let l:Integer=self.stGroup.startProp.lower in let u:UnlimitedNatural=self.stGroup.startProp.upper in if v="1" then l = 1 and u = 1 else if v="*" then l=0 and u =* else if v="0..1" then l=0 and u=1 else if v= "0..*" then l=0 and u=* else if v="1..*" then l=1 and u=* else parse(v) endif endif endif endif endif}
EndMultMap {CompartMap} attrName :String ="upper, lower"
StartAggrMap {CompartMap} attrName :String ="aggregation"
StartMultiplicity {EdgeCompartment} caption :String ="Start multiplicity" inputContr :InputContrType =Combobox position :LinePosition =start itemList :Set(String) =set{"1","0..1","0..*", "1..*"} compartNo :Integer =3 value :String
EndAggregat {EdgeCompartment} caption ="End aggregation" inputContr =Combobox position =end itemList =set{"none"," shared", "composite"} isVisible =false value :String
StartAggregat {EdgeCompartment} caption :String ="Start Aggregation" inputContr :InputContrType =combobox position :LinePosition =start itemList :Set(String) =set{"none"," shared", "composite"} isVisible :Boolean =false value :String

Fig. 24 Mapping specialization for the association fragment of EMOF class editor

the compartment is entered via a checkbox (and thus can have only the values "true" and "false"), UE can uniquely interpret the entered value as a Boolean type. One more case is when only literal names of an Enumeration type can be entered via a listbox having an appropriate item list, then they can be easily converted to literals of this type themselves. But, if the developer is unsure, an explicit OCL constraint may be used anyway. In the example in Fig. 23, the default transformation rules apply in all cases (the attributes have either String or Boolean type), so no explicit OCL is shown. But in the next fragment (Fig. 24) for one case an explicit OCL is required.

Figure 24 shows the mapping specialization for the EMOF class editor association fragment in Fig. 20 (as in the attribute case, mappings for all compartments are not shown, others are similar). Again the specialized mapping target metamodel is the same UML fragment in Fig. 22. The association compartment grouping into start and end groups is very adequate for mappings as well, since in the UML metamodel each of the association ends is represented by a Property instance attached under the corresponding Class. Therefore, the group mappings AssocStartMap and AssocEndMap both have links to the Property class. Since the Property class is used as a mapping link endpoint both for attributes and associations, it has two formal superclasses from the UMM—CompRowClass and LineEndClass. The association from Class to Property redefines three associations from UMM as well (for attribute row and both association ends). Only the association *name* mapping is related directly to the Association class in UML, and others are related to one of the groups. The value conversion from compartments to ASMM class attributes is supported by default facilities (as it is for attributes in Fig. 23), except one case – the association multiplicity (both at start and end, in Fig. 24 only the EndMultMap is detailed). The standard string values for multiplicity are "1," "0..1," "0..*," "1..*," but in abstract syntax they are coded by two attributes *lower* and *upper* of Property (in full UML, the coding is even more complicated, these attributes are derived ones, but we ignore this here). Therefore, the EndMultMap class references two attributes and use an OCL constraint specifying how exactly the values are mapped (the constraint in Fig. 24 supports only the standard values, but it can be extended to the general case). In addition, we remind that in UML the graphical and semantical start/end concepts are different for roles and multiplicities.

In addition, we have to extend the definition of UE for the mapping support. Some comments on the UE functionality were already made when UMM was explained, but here we give a more thorough summary. The general setting here is very simple—when a new graphical syntax element is created by the editor in a diagram, a new set of the corresponding mapping class instances is created as well. In addition, the relevant ASMM class instances are created as well. If an existing diagram element (compartment) is modified, the relevant modification is performed also in the corresponding ASMM class (on the basis of existing mappings and their links). UE has to perform all these tasks at the UMM instance and link level. There are no problems in this respect when UE processes the instances of UMM classes related to graphical editor or mapping since only attributes already present in UMM are used in a specialization (this permits to treat a specialized class instance as the corresponding UMM class instance as well). But ASMM class templates in UMM have no attributes at all—it is impossible to predict what attributes will be present in a specialized ASMM class taken, for instance, from a UML metamodel fragment. Some comments on this issue were already given when discussing the *attrName* feature in the extended UMM. The solution is that UE uses a more complicated internal coding for ASMM template class instances when attributes have to be added to them. Later on, these coded instances are interpreted in a uniform way as true instances of the specialized ASMM classes (in fact, predefined abstract syntax classes). Therefore, UE internally uses an extended UMM runtime version – these extensions are not relevant for the UMM specialization developer, see more in Sect. 12. Thus, the set of graphical diagrams in a project and the abstract syntax model is kept synchronized by UE during the editing. For our class diagram examples, the class attribute editing case is quite straightforward in this respect. But for associations the situation is slightly more complicated since UE has to attach the created Property instances (in fact, their coded "images" according to UMM) for association ends under the corresponding Class instances. Since UE knows from which class node to which one the association was drawn, the corresponding links to mapping class instances (existing also in the UMM, not only in the specialization) permit to locate the UML Class instances (in fact, their images in UMM) under which the new Property instances must be attached.

The extended UMM permits to extend the general functionality of UE in some aspects. First, the default graphical copy/paste facility in diagrams can be easily extended to a "semantic copying" where all graphical copies of a node reference the same mapping instance and consequently the same abstract syntax instance. In this situation, the mapping becomes many-to-one, and multiplicities in the UMM in Fig. 21 must be modified accordingly. Such a semantic copying is typically supported in all industrial UML editors. In addition, a very simple model tree support for a project could be easily added, including also the relevant abstract syntax elements. For this, the UMM in Fig. 21 has to be extended by three classes, which can be specialized to specify among other things that UML Class instances should be shown in the tree (not only the diagrams), and also under each class its properties should be shown. Thus, a functionality typical to standard UML class editors based on abstract syntax could be supported in our approach to a certain degree.

12 Implementation principles

Sections 6–11contain a sufficiently detailed informal description of the proposed functionality of UE for graphical editors from the editor user's point of view. Thus, a sort of requirements for the UE implementation are already present. The goal of this section is to provide proposals for adequate structuring of UE into components and some design patterns which would significantly simplify the UE implementation. All this is based on the experience of IMCS UL team in building the existing TDA platform.

First we want to recall that on the basis of the Type metamodel discussed in Sect. 3 (in fact, a certain extension of it – tool definition metamodel) the TDA platform [6,7,24,26,27,63] for graphical editor building has been implemented and used in practice for various graphical DSL

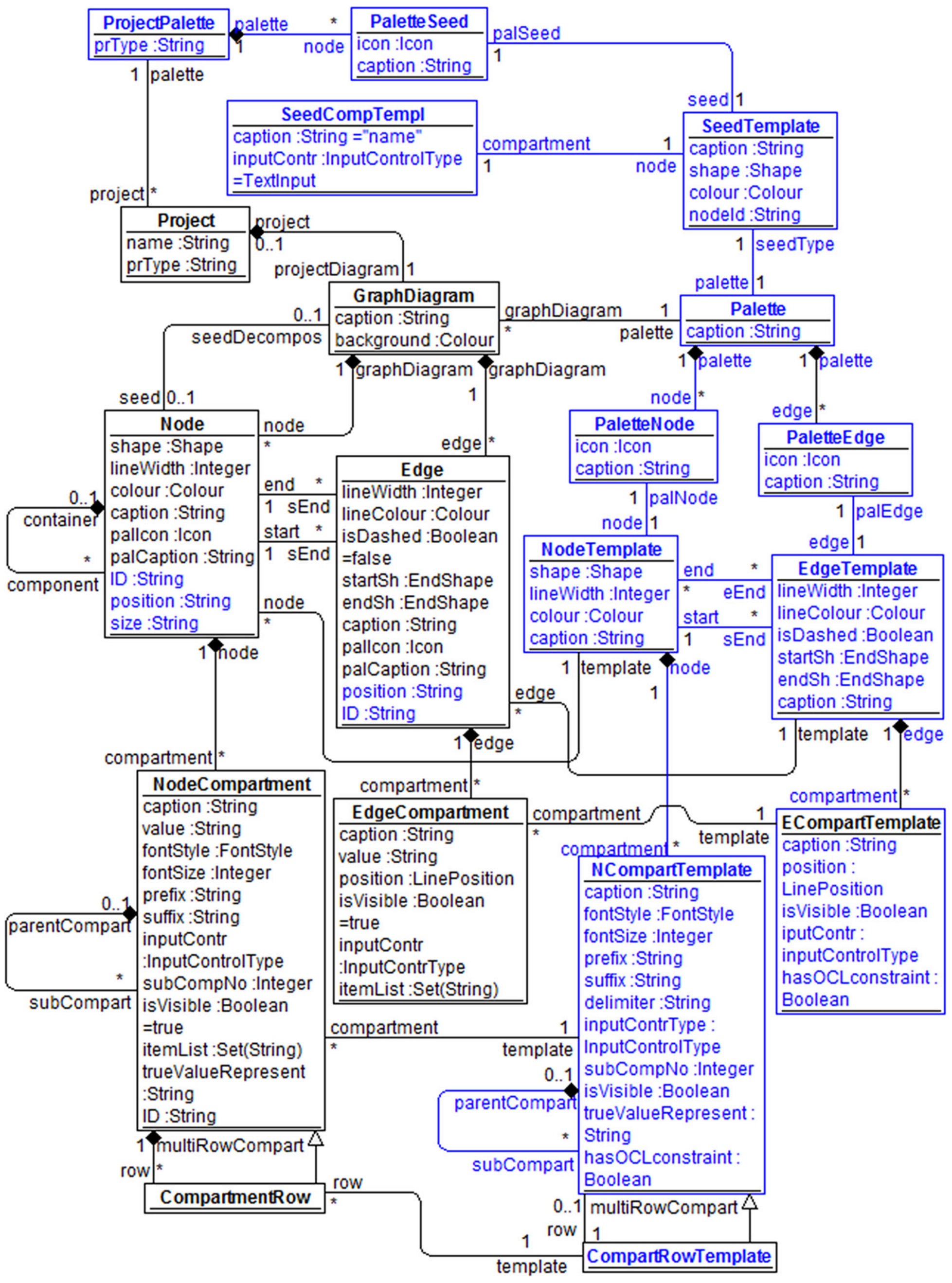


Fig. 25 Metamodel for palette tree

support. The platform contains the Main engine—a universal interpreter for a tool definition (an object model of the Type diagram) and several functional engines, such as Graph diagram (graphical presentation) engine and Dialog engine. The Graph diagram engine draws and modifies a diagram, supports an advanced layout algorithm, recognizes user events in a diagram and notifies the Main engine on "logical" events, e.g., a new node creation request. The Dialog engine builds forms for compartments and accepts user input; however, the logical processing of user input is done by the Main engine. A similar architecture could be used for implementing the UE associated with UMM for editor definition. The UE would consist of several components as well—the Main engine (ME) managing diagram projects, diagrams and diagram elements (all at UMM instance model level), UMM Specialization engine (SE) which navigates the current specialization of UMM and manages its instance set and the Graph Diagram engine (GDE) and the Dialog engine (DE) with a functionality similar to those in TDA. A new component required is the OCL constraint evaluator—but such OCL engines exist for several model repositories (see, e.g., [64], [65]). The main difference from TDA is the Specialization engine (SE) which has to process the specialized metamodel and provide the specific information in it (the specialized attribute values and redefined associations) to other engines in a generic way consistent with UMM. The only reasonable solution for UE is to store the runtime instances of diagrams and their elements in the current project according to the given specialization of UMM (SMM for short). This is done by SE in a project Repository (PR) according to SMM. In order to make other components of UE (ME, GDE and DE) independent of a specialization, the Specialization engine has to provide a temporary copy TR of the project Repository according to the original UMM. The copying process is simple—specialization instances are also instances of UMM classes, and attributes are the same. And in the temporary repository links for the redefined associations can also be interpreted as links for the original ones. Then, other engines can work in TR only in terms of UMM, but the Specialization engine provides the initial content of TR (for an existing project). In order to ease the synchronization between PR and TR, each UMM class at runtime is extended by a hidden ID attribute, and UE assigns a unique value to it at an instance creation. When an instance set creation (or modification) according to a user request is completed by UE, it notifies SE on new/modified instances via their Id value list. Then, SE can integrate these instances as proper SMM instances in PR. Modified instances are completely identified by the ID value (temporarily kept by SE also in PR). The class of a new instance created in TR is identified by its *caption* attribute to be explained later. The OCL expression evaluation is done by SE only in PR (upon requests by UE), since these expressions are fully based on SMM.

Now we explain how the minimal knowledge on SMM content—the class names, the fixed attribute values and the used redefined associations can be passed to UE in a usable way. In addition, the proposed solution helps to check that the information on the given specialization is sufficient for UE to function in a way specified in the previous sections. This is especially critical in the cases when new elements are to be created. The main idea behind this is that a new UMM instance creation by UE is always related to a palette. The palettes and dialog schemas for elements are created from specializations. There will be a language developer mode in the editor workbench, where one or more UMM specializations will be built as standard class diagrams, 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. The usage of such templates for the specification of a model fragment to be created is one of typical design patterns in model transformations. The palette tree is built according to a metamodel, a fragment of which is shown in Fig. 25. This fragment corresponds to the main elements of UMM for editors in Fig. 17 (groups for edge compartments are not shown). The metamodel is used only internally at runtime by UE and SE; therefore, the editor-related classes contain the technical attributes (ID, position, etc.). But a feature has to be explained here more in detail. All classes now have the *caption* attribute—this attribute was present already for most classes of UMM in Fig. 15. In UMM specializations, this attribute was set to a specific value in order to identify uniquely what diagram element kind the user currently is watching (e.g., in a dialog form). Therefore, its value, naturally, is unique for different subclasses of a UMM class. Here the *caption* attribute internally has a more formal role as well—it helps SE to identify the proper specialized class of a UMM class instance created by UE/ME. For classes where a visible caption is not needed, the value of this attribute is set internally to the specialized class name at the palette tree creation. This principle works well also for the extended version of the metamodel supporting the mappings. There is no such built-in identification for redefined role names in SMM, but SE can uniquely infer these names from role names in UMM when classes have been identified. This is because in cases where more than one association links two classes in UMM, the redefined role names must be different also in a correct specialization.

Figure 25 shows that the template structure in the palette tree replicates the required structure of diagram elements. 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. The internal links to be created also have the same role names as in UMM. Thus, the task of ME is to build in TR a complete instance replica of the relevant palette tree fragment when the user has requested a creation of a new diagram element. This principle naturally extends also to the case when along with diagram elements their mappings to abstract syntax must be created. Associations from template elements to the corresponding diagram elements are included in the metamodel to make this replication process easier (they are not maintained in the resulting model in TR).

Now some extended comments on ASMM class templates in UMM. These classes have no attributes at all in UMM; therefore, ME cannot directly add attributes to a new instance of such class template. To solve this, in the runtime UMM these classes are extended by a link to an internal Attribute class with two string-typed attributes *name* and *value*. ME

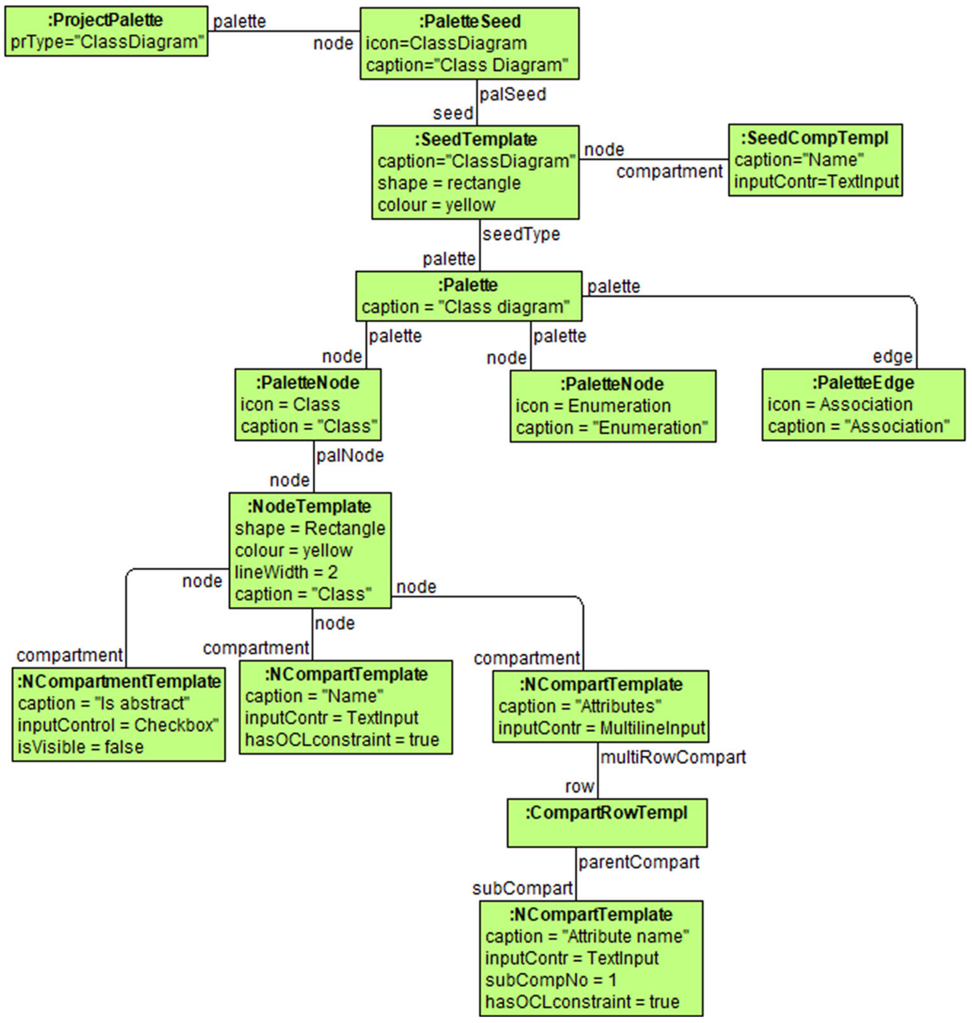


Fig. 26 Fragment of palette tree for class attributes

creates in a uniform way a new instance of Attribute class and links it to the instance of class template when logically a new ASMM class attribute has to be created according to a mapping for a new compartment instance in the graphical syntax (using the mapping templates included in the palette tree as well). When the user action processing is completed by ME, it notifies the SE on the new instances created (by ID list), including also these ASMM template class instances. SE can uniquely convert in PR these instances to true ASMM class instances (e.g., Class or Property). The relevant class name is coded in the class *caption* attribute, and linked Attribute instances are converted to true attributes of the ASMM class. The interpretation of string values as values of the corresponding type occurs at this moment (using an OCL expression when it is provided in the specialization). A similar temporal internal coding can be used also to hold links for the redefined UMM associations related to ASMM template classes.

Figure 26 shows a fragment of palette tree for the class diagram editor related to the attribute part.

Thus, the proposed implementation schema for a specialization based editor workbench is expected to require not very great effort since several components of UE (GDE, DE) could be reused from the existing TDA. It should be noted that ME and SE are true model transformation tasks. ME is a set of in-place transformations based on UMM (with a functionality similar to parts of TDA). The only completely new component is the SE for synchronizing the model in PR with the UMM based model in TR and performing some general management.

Now about the DSL developer workplace—its goal is to create a specialization of the given UMM. This workplace could be implemented on the basis of an appropriate open-source UML tool in Eclipse, e.g., Papyrus [66]. Then, one of the existing OCL interpreters, e.g., Eclipse OCL [64] could be used as well. All this is possible since metamodel specialization is completely based on standard UML features. However, the usage of standard UML tools (class diagram editor) for the DSL tool definition has a problem with prompting the possible UMM elements to be used in a specialization, etc.

Obviously, for defining DSL tools a specialized class diagram editor with advanced prompting facilities is required. Certainly, these prompting facilities will depend on the features of the given UMM. It is not difficult to get assured that such specialized class diagram editor can be built as a specialization of the editor definition UMM described in Sect.8. To implement such bootstrapping process, the initial version of this editor building tool must be created using a standard UML tool.

We are now in the process of development of a new specialization based platform for graphical DSLs at IMCS UL (according to the principles described above), using the experience and components the IMCS UL team has from the previous platform—TDA based on metamodel instantiation. From the end user point of view, the DSL tools created by the new approach, e.g., the EMOF level class diagram editor will look very similar to the existing TDA. The user interface for the desktop version will be the same as in Fig. 10. The user interface for the web version will be similar to that described in [67,68].

13 Summarized comparison of the specialization approach to instantiation approaches

In this section, we provide a summary of the comparison of our proposed metamodel specialization approach to the metamodel instantiation approaches analyzed in the paper. It will be done for the main two tasks discussed in the paper—graphical language syntax definition and building editors for these languages. There are two basic aspects in this comparison—the expressivity of the approach (what can be done in principle) and usability (how easy it is to do this for an end user).

Generally speaking, the formal expressive power of both approaches is equal—you can create as many subclasses of a UMM metamodel class (and with the same names) as instances (at M1 layer) of a metamodel class (at M2 layer). Only in the case of specialization we stay at the same layer. Similarly, the same principal possibilities are for class attributes and associations—except that for instantiation you can create as many new attributes and associations as you like, while for specialization they must be redefinitions of the existing ones in UMM. However, there may be differences in details. The situation is different for the TDA approach, since there the metamodel is already at M1 and its instantiation creates object diagrams.

However, in practice the styles of "start" metamodels are different. For specialization, we try to select the UMM as complete as possible for covering all features of the chosen family of tasks. Thus, for graphical syntax definition (see Sect. 7, Fig. 15) we have chosen the UMM covering all features of true graph diagrams, consisting of nodes and edges with arbitrary text structure and with node nesting supported. Since we do not use the domain (abstract syntax) model as a basic element, the concrete textual syntax is defined with any required detail using compartment structuring. Therefore, a complete set of required OCL constraints can be added to a UMM specialization, and these constraints can involve several related diagram elements (see Sect. 7). The basic style elements fixed in the corresponding graphical syntax specification are also included, with sufficiently rich sets of literals for enumerable types. Thus, we can assert that graphical syntax of any language based on true graph diagrams can be defined by a UMM specialization. But for example, the UML sequence diagram definition would require an extension of our diagram definition UMM.

The "classic" instantiation-based approaches applicable for graphical syntax definition such as Eclipse GMF and its derivatives (see Sect. 2) or OMG DD (see Sect. 4) do not have the syntax definition as the main goal. Thus, the main goal of GMF is the editor definition, but the main real goal of DD is the diagram interchange definition. In addition, they are strictly based on the domain model of the language (to which the main OCL constraints are attached) and on some sort of mapping the graphical syntax to this model. Therefore, the expressivity of the approach to a great degree depends on the mapping definition facilities used. In addition, the metamodels of graphical syntax to be instantiated in these approaches are significantly more general than our UMM; therefore, they cover a larger variety of diagrams but require more effort to create a specific diagram definition. The instantiation-based TDA (see Sect. 3 for details) is in a different position—it has a Type metamodel version (see Fig. 8) just for the graphical syntax definition, but due to one-layer technique it requires some non-UML semantics for the resulting object diagram, and not all constraints can be defined.

Now on the other task—the graphical editor definition for a language. The expressivity of the approach here depends both on the editor definition features included in the metamodel and possibilities of the runtime support in the platform. Typically the features supported by the runtime are all included in the corresponding platform metamodel (to be instantiated or specialized) – we call them standard features of the platform. In addition, usually there are default support features for diagram management not to be configured via the metamodel.

However, in practice graphical tool building for a language typically requires some specific features to be implemented by extending the standard runtime. And platforms typically have such extension possibility. Thus in Eclipse GMF (Sect. 2), there are broad possibilities to implement such extensions naturally in Java; however, they require a deep knowledge of the internal model structure and existing runtime libraries in Eclipse. In TDA platform (Sect. 3), there is a special concept of an extension point in the editor-oriented metamodel, where custom transformations in Lua/lQuery language can be invoked. It should be noted that the standard facilities for editor definition in TDA (e.g., toolbar elements, context menus) are richer and easier configurable than in Eclipse. However, the one-layer metamodel in TDA with its mix of elements and their types frequently makes such transformation development complicated—model transformations typically are oriented toward model-metamodel separation.

Now about the situation in our metamodel specialization approach for editor building. Since the UMM for editor definition (see Sect. 8, Fig. 17) is a direct extension of the UMM for language definition, the general capabilities of specialization, including OCL constraints, seem to be sufficient for defining any standard feature of editor (similarly to the language definition case). However, custom features extending the capabilities of UE (which supports the given UMM) would be required anyway. Since our UMM is directly oriented toward graphical syntax, the need for such extensions could be less than, e.g., in GMF. Many custom editing features directly related to metamodel elements can be specified simply as additional OCL constraints. However, for features related to some external elements, such as diagram export in various formats and custom context menu items. OCL constraints may not be sufficient. A possible solution could be to use code fragments in some transformation language instead of OCL as our extension points. These transformations could also access directly the runtime repository API (where UE stores the models, see Sect. 12). This possibility most probably would be sufficient for most editor building cases. An efficient management of extension points may require adding a couple more editor-specific attributes to diagram element-related classes of UMM.

Finally, on the comparison of usability. For the sake of simplicity, we have chosen the flowchart diagram as the common test case. But a similar situation would be for a more complicated diagram as well. The flowchart example—for both language definition and editor definition—shows convincingly that the specialization approach is much simpler both for development and understanding. For flowchart syntax definition, compare Fig. 16 (Sect. 7) using the specialization approach to the corresponding figures for other approaches. Thus for GMF as a language definition, you have to create (or understand) Figs. 2, 3 and 5 (Sect. 2). For TDA, the corresponding definition is in Fig. 9 (Secti. 3)—though not very large, it is based on non-UML semantics for several elements in this diagram; in addition, not all semantic constraints can be specified there. For OMG DD, Fig. 11 (Sect. 4) presents only the DI model part of the solution—it has to be accompanied by a mapping of this model to the true graphics—the DD metamodel.

A similar situation is with the flowchart editor definition. Fig. 18 (Sect. 9) shows the complete editor definition via specialization. But the Eclipse GMF solution requires Figs. 2, 3, 4 and 5 (Sect. 2) plus the generation model not shown in the paper. Though the class diagram editor example (Sect. 10) is much more complicated, the essential fragments of this definition via specialization in Figs. 19 and 20 are still very readable.

One more aspect of comparison is the maintainability (ease of modification/extension) of solutions. The specialization approach requires only one model to be modified—the specialization for language or editor (including its OCL constraints). If the mapping to abstract syntax is defined as well, then only this specialization should be modified (it includes the editor definition as a submodel). There are no related models which should be modified as well—as it is, for example, in GMF or OMG DD. Thus in GMF, you have to modify consistently the domain model, graphics model and mapping model (frequently also the tooling and generation model); therefore, this modification is very error-prone [18].

14 Conclusions

A unified approach for graphical diagrammatic language syntax definition and graphical editor building has been proposed in this paper. The approach is based on a Universal Metamodel (UMM) which is then specialized for a concrete language. We want to emphasize once more that the whole approach uses metamodel specialization technique, but not the metamodel instantiation mainly used so far. The most popular use cases of the traditional instantiation-based approach for language and editor definition have been briefly analyzed in Sections 2–5, with their advantages and drawbacks discussed. Sections 6–12 provide a detailed description of the proposed specialization approach.

Section 13 provides a summary of comparison of the metamodel specialization approach to typical metamodel instantiation approaches. This comparison clearly shows that the specialization approach is simpler and easier to use for the definition of graphical modeling language syntax and their editors. This is especially visible in the case of simpler languages (such as the flowchart example is), but the class diagram example shows that the same would be true for more general domain-specific modeling languages as well.

Now about our future plans. We are in the process of development of a new specialization based platform for graphical DSLs, according to the principles described in Sect. 12. The goal is to create a DSL platform with a functionality similar to the existing TDA, but much more user friendly due to the specialization approach. Besides that, the platform should be accessible via web browser. In order to check the web access possibility for such a platform, one of our team members has performed a successful experiment (however, for the instantiation-based approach) [67,68]. The main advantage here is the possibility for several users to access the same diagram, with an automatic synchronization at updates. This will solve the problem of team development mode for DSLs, similarly to Eugenia Live and Collaboro (see Sect. 5).

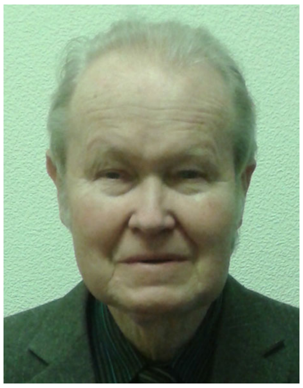
The main proposed application area for the specialization approach is the graphical DSL and their tool definition, discussed in this paper. However, we believe that the approach can be applied to a much broader areas. A very promising use case could be the definition of user interfaces for web-based information systems. Currently, there is an instantiation-based trial to use metamodels for user interface definition—the IFML standard by OMG [69–71]. We hope that the application of the specialization approach could provide advantages in this case as well, especially by extending the variability of supported user interface kinds. Another possible use case could be the ontology building based on specialization. Ontology specialization could be used as a new facility for large ontology structuring, with the main goal to represent complicated ontologies in an easy understandable way.

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

References

1. Object Management Group. Meta Object Facility (MOF) Core Specification – Version 2.5 – formal/2014-06-05 (2015)
2. Object Management Group. Unified Modeling Language (UML) – Version 2.5 – formal/2015-03-01 (2015)
3. Object Management Group. Diagram Definition (DD) – Version 1.1 – formal/2015-06-01 (2015)
4. Graphical Modeling Framework (GMF) Tooling. http://eclipse.org/gmf-tooling/
5. Eclipse Modeling Framework (EMF). https://projects.eclipse.org/projects/modeling.emf/
6. Barzdins, J., Rencis, E., Kozlovics, S.: The Transformation-Driven Architecture. In: Proceedings of DSM'08 Workshop of OOPSLA 2008, Nashville, Tennessee, pp. 60 – 63, University of Alabama at Birmingham (2008)
7. Sprogis, A.: Configuration Language for Domain Specific Tools and its Implementation. Ph.D. thesis (in Latvian), University of Latvia, Riga (2013)
8. Kalnins, A., Barzdins, J.: 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, pp. 103–112 (2016)
9. Graphical Editing Framework (GEF). http://www.eclipse.org/gef/
10. Obeo Designer: Domain Specific Modeling for Software Architects. http://www.obeodesigner.com
11. Juliot, E. Benois, J.: Viewpoints creation using Obeo Designer or how to build Eclipse DSM without being an expert developer? Obeo Whitepaper. http://spotidoc.com/doc/197222/
12. Sirius overview. http://www.eclipse.org/sirius/overview.html
13. Acceleo – Eclipsepedia. http://wiki.eclipse.org/Acceleo
14. Kolovos, D., Rose, L., et al.: Taming EMF and GMF using Model Transformation. In: Petriu, D., Rouquette, N., Haugen, O. (eds.) Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), LNCS, Vol. 6394, pp. 211–225. Springer (2010)
15. Kolovos, D., Rose, L., et al.: Eugenia: towards disciplined and automated development of GMF-based graphical model editors. SoSyM **16**(1), 229–255 (2017)
16. Kolovos, D., Roze, l., Garcia-Dominguez, A., Paige, R.: The Epsilon Book. http://www.eclipse.org/epsilon (2017)
17. Kouhen, A., et al.: Evaluation of modeling tools adaptation. HAL archives (2012). https://hal.archives-ouvertes.fr/hal-00706701/file/Evaluation_of_Modeling_Tools_Adaptation.pdf
18. Wienands, C. Golm, M.: Anatomy of a visual domain-specific language project in an industrial context. In: Proceedings of 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009), LNCS, Vol. 5795, pp. 453–467. Springer (2009)
19. IBM Rational Software Architect Designer RSA). https://www.ibm.com/support/knowledgecenter/SS8PJ7/rsa_family_welcome.html
20. Kalnins, A., Vilitis, O., Barzdins, J., et al.: Building Tools by Model Transformations in Eclipse. In: Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyvaskyla University Printing House, pp. 194–207 (2007)
21. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. In: Proceedings of Model Driven Architecture: European MDA Workshops: Foundations and Applications, LNCS, Vol. 3599, pp. 62-76. Springer (2005)
22. Vilitis, O.: Metamodel-based transformation-driven graphical tool building platform. GlobeEdit (2015)
23. MOLA Home, IMCS University of Latvia. http://mola.mii.lu.lv/index.html
24. Barzdins, J., et al.: GrTP: Transformation Based Graphical Tool Building Platform. In: Proceedings of MDDAUI'07 Workshop of MODELS 2007, Nashville, Tennessee, USA, CEUR Workshop Proceedings, Vol. 297, 4 pp. (2007)
25. Liepins, R.: Library for model querying – lQuery. In: Proceedings of 2012 Workshop on OCL and Textual Modelling (part of Models 2012), ACM Digital Library, p. 6 (2012)
26. Sprogis, A.: The configurator in DSL tool building. Comput. Sci. Inf. Technol. Sci. Pap. Univ. Latv. **756**, 173–192 (2010)
27. Sprogis, A., Barzdins, J.: Specification, configuration and implementation of DSL tool. In: Frontiers of AI and applications, Databases and Information Systems VII, Vol. 249, pp. 330–343, IOS Press (2013)
28. Barzdins, J., Barzdins, G., Cerans, K., Liepins, R., Sprogis, A.: UML style graphical notation and editor for OWL 2. In: Proceedings of Perspectives in Business Informatics Research (BIR 2010), LNBIP, Vol. 64, pp. 102–113. Springer (2010)
29. Barzdiņš, J., Barzdiņš, G., Cerans, K., Liepiņš, R., Sprogis, A.: OWLGrEd: a UML Style Graphical Editor for OWL. In: Ontology Repositories and Editors for the Semantic Web, Proceedings of the 1st Workshop on Ontology Repositories and Editors for the Semantic Web, Hersonissos, Greece, CEUR Workshop Proceedings, Vol. 596, p. 5 (2010)
30. Cerans, K., Liepiņš, R., Ovcinnikova, J., Sprogis, A.: Advanced OWL 2.0 Ontology Visualization in OWLGrEd. In: Frontiers of AI and Applications, Databases and Information Systems VII, Vol. 249. pp. 41–54, IOS Press (2013)
31. Liepins, R., Grasmanis, M., Bojars, U.: OWLGrEd ontology visualizer. In: Proceedings of the International Semantic Web Conference, Developers Workshop 2014 (ISWC-DEV'2014), CEUR Workshop Proceedings, Vol. 1268, pp. 37–42 (2014)
32. OWLGrEd home, http://owlgred.lumii.lv/

33. Barzdins, J., Cerans, K., Grasmanis, M., Kalnins, A., Kozlovics, S., Lace, L., Liepins, R., Rencis, E., Sprogis, A., Zarins, A.: Domain specific languages for business process management: a case study. In: Proceedings of 9th OOPSLA Workshop on Domain-Specific Modeling, Orlando, USA, October 2009, pp. 34–40 (2009)
34. GradeTwo Tool. http://gradetwo.lumii.lv/
35. Eisenberg, J.: SVG Essentials. O'Reilly Media, Sebastopot (2011)
36. Object Management Group, Meta Object Facility (MOF) 2.0 Query/ View/Transformation Specification – Version 1.2 – formal/2015-02-01 (2015)
37. Elaasar, M., Labiche, Y.: Diagram definition: a case study with the UML class diagram. In: Proceedings of the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011), LNCS, Vol. 698, pp. 364-378. Springer (2011)
38. Fouché, A., Noyrit, F., Gérard, S., Elaasar, M.: Systematic generation of standard compliant tool support of diagrammatic modeling languages. In: Proceedings of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), pp. 348–357, IEEE (2015)
39. Costagliola, G., Deufemia, V., Polese, G.: A framework for modeling and implementing visual notations with applications to software engineering. ACM Trans. Softw. Eng. Methodol. **13**(4), 431–487 (2004)
40. Rekkers, J., Schurr, A.: Defining and parsing visual languages with layered graph grammars. J. Vis. Lang. Comput. **8**(1), 27–55 (1997)
41. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object oriented and rule-based design of visual languages using tiger. In: Proceedings of GraBaTs'06, Electronic Communications of the EASST, p. 12 (2006)
42. Taentzer, G., Crema, A., Schmutzler, R., Ermel, C.: Generating domain-specific model editors with complex editing commands. In: Proceedings of AGTIVE 2007, LNCS, Vol. 5088, Springer, pp. 98–103 (2007)
43. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003), LNCS, Vol. 3062, pp. 446-453. Springer (2004)
44. Rath, I., Varro, D.: Challenges for advanced domain-specific modeling frameworks. In: Proceedings of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, p. 4 (2006)
45. Visual Automated Model Transformations (VIATRA2), GMT subproject, Budapest University of Technology and Economics. http://dev.eclipse.org/viewcvs/indextech.cgi/gmthome/subprojects/VIATRA2/index.html
46. Graphiti Home. http://www.eclipse.org/graphiti/
47. Rose, L., Kolovos, D., Paige, R.: EuGENia Live: a flexible graphical modelling tool. In: Proceedings of XM'12, ACM Digital Library, pp. 15–20 (2012)
48. Reid, J., Valentine, T.: JavaScript Programmer's Reference. Apress, New York (2013)
49. Izquierdo, J.C., Cabot, J.: Collaboro: a collaborative (meta) modeling tool. PeerJ Computer. Science **2**, e84 (2016)
50. Cook, S., Jones, G., Kent, S., Wills, A.C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional, Boston (2007)
51. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, Hoboken (2008)
52. Bardohl, R., Ermel, C., Weinhold, I.: GenGED – A visual definition tool for visual modeling environments. In: Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003), LNCS, Vol. 3062, pp. 413–419. Springer (2004)
53. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Sci. Comput. Program. **44**(2), 157–180 (2002)
54. Vangheluwe, H., de Lara, J.: Domain-specific visual modelling in AToM3. Proceedings of DSM 04, p. 8 (2004). http://www.dsmforum.org/events/DSM04/papers.html
55. Pierre, S., et al.: A family-based framework for i-DSML adaptation. In: Proceedings of 10th European Conference ECMFA 2014, LNCS, Vol. 8569, pp. 164–179. Springer (2014)
56. Bruck, J., Damus, C.: Creating Robust Scalable DSLs with UML, eclipsecon 2008 tutorial. https://www.eclipsecon.org/2008/indexf901.html?page=sub/&id=172 (2008)
57. Atkinson, C., Kuhne, T.: Concepts for comparing modeling tool architectures. In: Briand, L., Williams, C. (eds.) Model Driven Engineering Languages and Systems: 8th International Conference (MODELS 2005), LNCS, Vol. 3713, pp. 398–413. Springer (2005)
58. Atkinson, C., Gerbig, R., Kuhne, T.: A unifying approach to connections for multi-level modeling. In: Proceedings of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), pp. 216–225, IEEE (2015)
59. Clark, T., Gonzalez-Perez, C., Henderson-Sellers, B.: A Foundation for Multi-Level Modelling. In: Proceedings of the Workshop on Multi-Level Modelling (MULTI 2014), CEUR Workshop Proceedings, Vol. 1286, pp. 43–52 (2014)
60. Atkinson, C., Kuhne, T.: Profiles in a strict metamodeling framework. Sci. Comput. Program. **44**(1), 5–22 (2002)
61. Atkinson, C.: Supporting and applying the UML conceptual framework. In: Proceedings of the Conference: The Unified Modeling Language (UML'98), LNCS, Vol. 1618, pp. 21–36. Springer (1998)
62. Atkinson, C., Gerbig, R., Kuhne, T.: comparing multi-level modeling approaches. In: Proceedings of the Workshop on Multi-Level Modelling (MULTI 2014), CEUR Workshop Proceedings, Vol. 1286, pp. 53-62 (2014)
63. Rencis, E., Barzdins, J., Kozlovics, S.: Towards open graphical tool-building framework. In: Proceedings of BIR 2011, pp. 80-87, RTU Press, Riga (2011)
64. Eclipse OCL (Object Constraint Language). https://projects.eclipse.org/projects/modeling.mdt.ocl
65. Dresden OCL. https://github.com/dresden-ocl
66. Papyrus Project in Eclipse. http://projects.eclipse.org/projects/modeling.mdt.papyrus
67. Sprogis, A.: DSML Tool Building Platform in WEB. In: DB&IS 2016 Proceedings, CCIS Vol. 615, pp. 99–109. Springer (2016)
68. Sprogis, A.: ajoo: WEB Based framework for domain specific modeling tools. In: Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IX, Vol. 291, pp. 115–125. IOS Press (2016)
69. Object Management Group. Interaction flow modeling language (IFML) Version 1.0 formal/2015-02-05 (2015)
70. Brambilla, M., Fraternali, P.: Interaction Flow Modeling Language. Model-Driven UI Engineering of Web and Mobile Apps with IFML. Elsevier, Amsterdam (2015)
71. Wazlawick, R.: Object-Oriented Analysis and Design for Information Systems. Modeling with UML, OCL and IFML. Elsevier, Amsterdam (2014)



Audris Kalnins is a part-time full professor at the University of Latvia and a senior researcher at the Institute of Mathematics and Computer Science (IMCS), University of Latvia. His research interests include model-driven development, model transformations, metamodeling, domain-specific languages and development tools for such languages. He holds Ph.D. and Dr. Habil. Degree in Computer Science from University of Latvia. He has lead the development of model transformation language MOLA and has been the principal investigator for the EU research project ReDSeeDS at IMCS.

Janis Barzdins is a Full member of the Latvian Academy of Sciences, a part-time full professor at the University of Latvia and a senior researcher at the Institute of Mathematics and Computer Science, University of Latvia. He heads the SOPHIS project "Ontology-based knowledge engineering technologies". His research interests include conceptual modeling, metamodeling, model transformations, model-driven engineering, tool-building frameworks. He is the author of more than 120 scientific papers in different areas of computer science, including algorithm theory, inductive inference, modeling technologies, model transformations, tool-building frameworks, etc.