DIAGRAM DEFINITION FACILITIES BASED ON METAMODEL MAPPINGS

Edgars Celms, Audris Kalnins, Lelde Lace

University of Latvia, IMCS, Riga, Latvia

{ Edgars.Celms, Audris.Kalnins, Lelde.Lace }@mii.lu.lv

The paper proposes a new technique for diagram definition in a generic modeling tool, which permits to build several diagrammatic presentations for one domain. The main idea of the proposed method is a mapping from presentation to domain part of a metamodel. The paper describes the semantics of mappings, using a fragment of UML activity diagram as a definition example. In conclusion suggestions are given how the approach could be applied in MDA context.

Introduction

Besides the traditional modeling tools built for a specific modeling notation such as object modeling in UML, business modeling etc. there is a significant niche for generic modeling tools where any modeling notation can be supported without programming in the traditional sense. Typically the modeling notation (language) in generic modeling tools is specified by means of a metamodel, which is then augmented by a tool specific annotation, markup etc, to define the actual tool functionality. The main application area for generic modeling tools is domain-specific visual languages for various industry domains. The current key players in this area are ISIS GME [1], DOME [2], MetaEdit [3] etc, which have gained certain maturity now.

The classical graphical modeling by means of sets of related diagrams can also be considered to be a domain-specific area, especially the business modeling, where there is no one leading modeling language, but a number of quite similar competing notations. UML with its Activity diagrams is just one of the possible notations there. Therefore the generic modeling approach is valid also for the domain of business modeling. This domain poses some specific requirements for the tool, the most important one being the necessity to represent the same domain concepts via several graphic notations simultaneously. The paper discusses the Generic Modeling Tool (GMT), developed by University of Latvia together with Exigen company, built especially for the abovementioned purpose. Namely the requirement for access to the same model data via several graphical notations demands a number of specific solutions for defining the relations between diagrams and domain metamodel (the diagram definition language), which can not be so easily accomplished in the well-known metamodel-based tools [1,2,3]. For example, there it is practically impossible to have some domain object represented as a symbol in one notation and as a line in another.

The main such idea is the mapping between the parts of the metamodel – the domain and the presentation part (the latter ones may be several). Some preliminary presentation of the approach has been given in [4,5], but this paper concentrates on precise definition of the mapping semantics, using UML and OCL. Though developed completely independently, the style of the semantics definition bears some similarity to the more theoretical paper [6], where the concept of set-theoretical relation between metamodel parts is used.

It should be noted that the requirement for alternative graphical notations is present in the UML itself (including the version 2.0) – interactions can be shown both as sequence and collaboration diagrams, there are alternative forms of showing action performers in activity diagrams. The paper will demonstrate the mapping idea on a small fragment of UML 2.0 activity diagram metamodel [7], finally showing how the same model data could be presented as ARIS eEPC [8] diagrams – the most popular business process notation.

Today the hottest area in UML related modeling is MDA [9]. The paper concludes with some suggestions, how the proposed approach can be used for this goal too.

Structuring of metamodels

Strictly speaking, a metamodel for a modeling notation such as UML describes only the domain concepts – the abstract syntax in other terms. But any modeling tool must manipulate also the elements of the diagrammatic presentation – the concrete graphical syntax. UML documents [7] does not specify that part of the metamodel, a very generic description of the presentation part is given in [10], however with another goal in mind – defining an easy interchange format for graphics. Therefore as a rule modeling tools internally use another presentation metamodel, and so it is in our approach. Similarly to [10], we assume a diagram to be a directed graph consisting of nodes (boxes) and edges (lines). The presentation metamodel is also the only natural place, where various constraints on diagram building can be easily specified.

In our approach any metamodel is built according to MOF standards – it is a class diagram using the syntax features permitted by MOF. The elements of a metamodel may be grouped into packages, so we can speak of **domain** and **presentation packages**. Fig. 1 shows the domain package for the UML 2.0 activity diagram (actually a small fragment of the original one in [7], but sufficient for demonstrating the ideas). It should be reminded, that the *Activity* class plays the role of the "domain diagram" – its instances correspond to instances of visible activity diagrams.

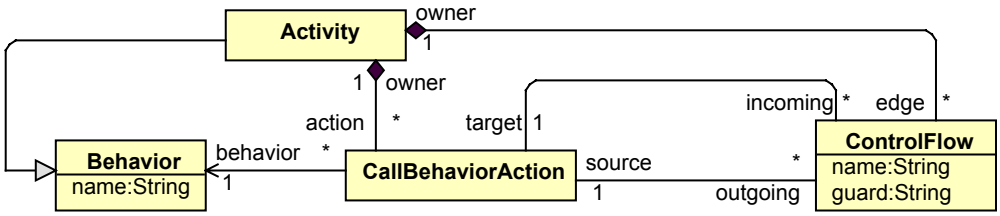

Figure 1. Domain package for UML activity diagram

To reflect the fact that any diagram is a graph, a special **DiagramCore** package is introduced which defines the general properties of a graph. Any specific presentation package inherits these properties from the core and may add some specific features required by the graphic syntax (actually the DiagramCore is more than a graph – it supports element nesting etc., but we omit this for simplicity). The DiagramCore elements also contain attributes characterizing their geometrical properties, but we ignore them here. Fig. 2 shows the DiagramCore package together with the presentation package for activity diagram.

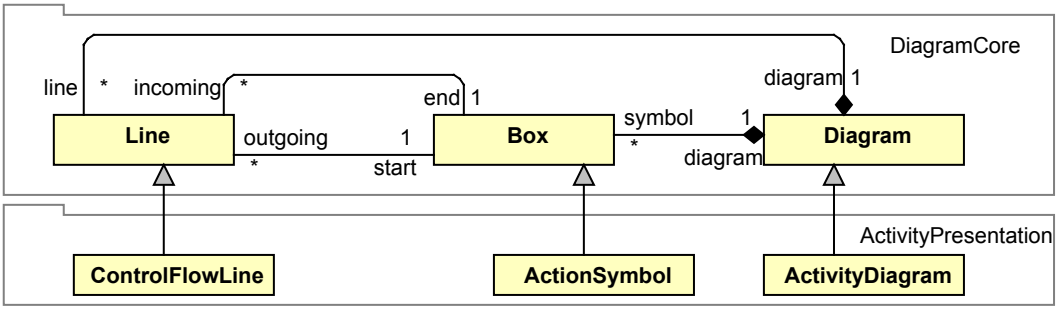

Figure 2. Diagram core and Activity presentation packages

Similarly, for any diagrammatic notation there is one domain package and one or more presentation packages, all of which inherit their essential properties from the DiagramCore. In a sense, it is an extension of the original domain metamodel, but with the domain packages left unmodified. The only elements, which will be added to the domain part of the

metamodel, will be some new associations – mapping associations, which are the basis of our approach and which will be discussed in the next sections.

One more consequence of the DiagramCore is that we can use a separate component – **Graphical Diagramming Engine** (GDE) – to support the graphical functionality of diagrams. This component is based upon complicated graph drawing algorithms [11] and implements all tool features, which can be expressed in terms of extended directed graphs. This way we isolate the pure graphical functionality of diagram building. GDE supports creating a new and opening an existing diagram and allows to perform all graphical operations with diagram elements (add, delete, move, etc.). GDE supports also automatic high quality layouts of a diagram and enables style modification of diagram elements.

General principles of mapping

The domain and presentation parts of the metamodel for a modeling notation must be linked together to define the real modeling functionality. In most cases, a class in the domain part corresponds to a class in the presentation part, but these correspondences may be also more complicated. The main facility for defining a relation between the domain and presentation parts of the metamodel is mapping.

In the simplest case, the **mapping** consists of a **class** in the **presentation** package, the corresponding **class** in the **domain** package and an **association** connecting them. It expresses the fact that as soon as there is a presentation class instance (e.g., action symbol) there must also be the corresponding domain class instance (CallBehaviorAction) and vice versa, and they must be linked by the association instance (link). Typical association multiplicities are 1 – 1 (for one graphical notation). The associations (called **mapping associations**), navigable to both ends, form the base for efficient data management in the Generic Modeling Tool.

However, there is more semantics related to a mapping. Thus, the action symbol must be in the diagram, which is mapped to the activity containing the action. Even more complicated rules constrain mappings for lines, where natural conditions tie a line mapping to its end box mappings. Thus, mappings form hierarchical structures, corresponding to basic diagrammatic constructs or patterns. Each such pattern corresponds to a **mapping type** in our approach. Some basic mapping types will be discussed in the next section. Each of these mapping types will have its **syntax** – the involved metamodel elements, and **semantics** – what constraints must be true for the mapping to be in place (or in other words, what must be done, if one of the mapping ends has changed).

The mapping semantics is based on the hierarchy of mapped elements. There is one common principle in this semantics, so called **scaffolding principle**, explained in Fig. 3. The scaffolding principle specifies how mappings must be consistent with the element hierarchy on both ends. The explanation of the principle to a certain degree reminds the relation principle in [6], but is simpler.

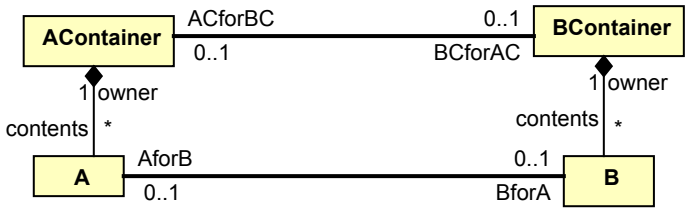

Figure 3. Scaffolding principle for mappings

Let *A* and *B* be classes in the presentation and domain packages respectively, involved in a mapping, and let *AContainer* and *BContainer* be their corresponding owners in the hierarchy (typically, a diagram and its domain equivalent), also having their own mapping.

Both mapping associations are displayed bold. These elements are assumed to be really present in the metamodel. Fig. 3 actually represents a general **scaffolding schema**.

In a totally correct model the mapping association multiplicities must be 1 – 1 (we consider here the one-one case, the one-to-many case is a completely different pattern). However, for an in-process model (being modified by the tool) some mapping instances may be temporary missing, therefore multiplicities in the schema are set to 0..1.

According to the principle, the following two constraints given by OCL expressions must always hold for the scaffolding schema involving the abovementioned mappings:

Context A **inv**:
BforA->notEmpty() implies owner.BCforAC = BforA.owner

and

Context B **inv**:
AforB->notEmpty() implies owner.ACforBC = AforB.owner

These constraints express the fact that *A* to *B* mapping is consistent with the corresponding container mapping – e.g., a symbol maps to an action in the right "domain diagram".

The most important condition for this schema is the **local completeness** for one container (diagram), which can be expressed by the following OCL constraint:

Context AContainer **inv:**
contents -> forAll (a | a.BforA->notEmpty()) and
BCforAC->notEmpty() and
BCforAC.contents -> forAll (b | b.AforB->notEmpty())

The constraint says that for this container all its elements are mapped (consistently with the container mapping) and, in addition, are mapped to all elements of the partner (domain) container – the mapping is complete both ways for the given container. Mapping for any diagram type will try to maintain its local completeness for the current diagram instance.

The scaffolding principle is the base for all mapping types to be discussed in the next section.

Mapping types for the activity diagram example

Now the mapping types to be used for the activity diagram example can be defined. The base for all other mappings is the **diagram mapping** – a special singleton mapping. Fig.4 shows the diagram mapping for activity diagrams. It says that each activity diagram instance must correspond to an *Activity* class instance, having a consequence that creating a diagram in the tool must invoke a new *Activity* instance creation.

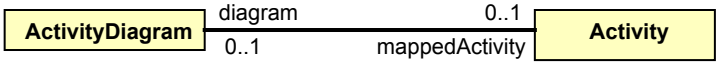

Figure 4. Diagram mapping

The simplest non-trivial mapping type is for a kind of diagram symbols to be mapped to a domain metamodel class. In our simplified activity diagram example there is only one symbol kind – *ActivitySymbol*, which must be mapped to the *CallBehaviorAction* in the domain. This mapping type will be named **1OT** (symbol to 1 Object Type).

Each mapping type defines its mapping schema, based on the scaffolding principle. The mapping schema contains a number of metamodel elements both from presentation and domain packages – the **mapping syntax**, which must be substituted by concrete metamodel elements when concrete mapping is defined. The **mapping semantics** is, firstly, inherited from the scaffolding principle (its constraints), and more constraints can be added for a mapping type definition. But a concrete mapping, as a rule, adds no OCL constraints (however, there is such a possibility in the modeling tool), so concrete diagram definitions

typically requires no explicit use of OCL and constraints inherited from mapping types can be implemented in a more efficient way by the modeling tool.

Fig. 5 shows the mapping schema for the mapping type 1OT. The mapping syntax (list of parameters) contains the classes *Diagram* and *Symbol*, which must be located in the metamodel presentation package, and the classes *DomainDiagram* and *DomainElement*, which must be in the domain package. The two associations (*Diagram* to *Symbol* and *DomainDiagram* to *DomainElement*) are also part of the syntax and must be found in the metamodel. The mapping association for diagrams must already be defined. But the mapping association for the *Symbol* (actually, for its real counterpart in the metamodel) must be specially created before a concrete mapping is defined.

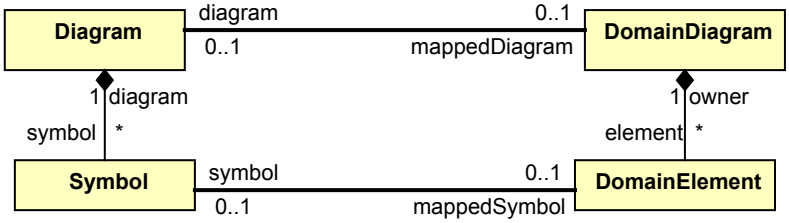

Figure 5. 1OT Mapping schema

The mapping type 1OT adds no new constraints to those inherited from scaffolding schema. Let us remind, that it requires also the local completeness condition to be true for a mapping to be complete. In practice, these constraints imply that creating a new symbol in a diagram means also the creation of the domain element – thus the "operational semantics" required by the tool is also defined by the mapping type.

There is also a variation of the mapping type 1OT named **1OTD** (Object Type with Definition), which adds one more class in the domain, linked by an association to the DomainElement), this additional class serves as a common "definition" for the domain elements. Namely the variation 1OTD is used for the activity example – see the concrete action mapping in Fig. 6, with *Behavior* in the role of the definition (the association *refinement* is explained below). Associations inherited from the DiagramCore here (and in the next figure) are shown directly between the presentation classes.

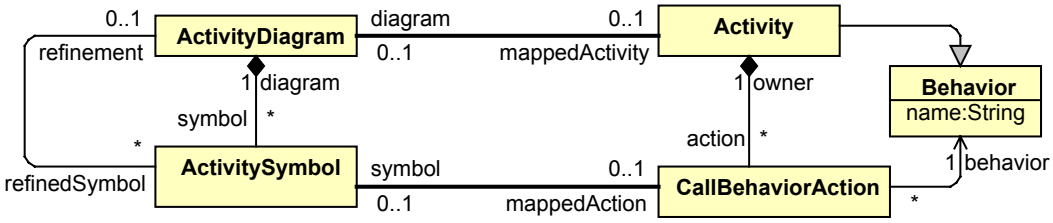

Figure 6. Action mapping according to 1OTD

We will demonstrate one more extension of the type 1OT – **1OTR** (1OT with **refinement**). An action referencing another *Activity* (not a simple *Behavior*) implies that the corresponding *ActionSymbol* must be displayed as refined (with the rake icon in it) and must support hyperlinking to the appropriate diagram. Due to restricted space the definition of 1OTR is not given, just its application is shown in the same Fig. 6. This extension requires new OCL constraints, which here will be demonstrated directly in the application (not in the schema definition, as it in fact is). The idea is that we select an association in the presentation package (with the role *refinement*), which makes the *ActionSymbol* to be refined. This association should be paired to an association in the domain between the relevant classes, in this situation the same *behavior* association may be used, in case if it leads to another *Activity* (as a subclass of *Behavior*). The same *refinement* association also enables hyperlinking to the appropriate diagram. 1OTR schema requires the following additional OCL constraints (here shown in the concrete context of Fig. 6)

Context ActionSymbol **inv**:
refinement->notEmpty() implies (mappedAction.behavior.oclIsTypeOf(Activity) and
refinement= mappedAction.behavior.diagram)

The next essential mapping type is for lines in diagram, which correspond to classes in the domain. This mapping type is named **L1OT**. To save the space, we again do not present separately its schema, but show its application for defining control flows in activity diagram. The basis again is the scaffolding schema, but a similar principle here has to be applied directly in a specific context, in order to specify that line ends (the boxes to which the line is attached) are mapped accordingly. Fig. 7 shows this definition.

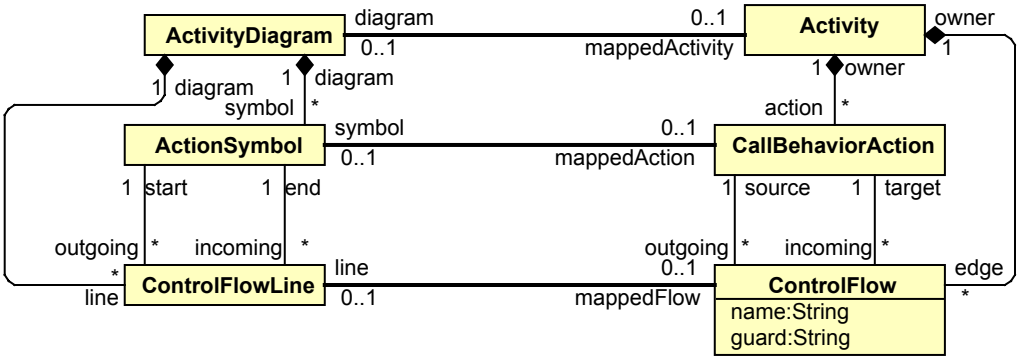

Figure 7. Control flow Line Mapping

A direct application of the scaffolding schema here links the line mapping to the diagram mapping, with the corresponding constraints inherited. But the action mapping is repeated here to specify the line ends. In this simple example there is only one symbol type, but if a control flow could be drawn between different symbol types (as it is in real activity diagrams), all these mappings had to be present in the definition. However, this is required only for the semantics definition, in real diagram definition facility only the relevant pairs of symbol types are to be listed (and also the relevant association roles in the domain), the corresponding OCL constraints are inherited from the L1OT schema (here shown only as its application in Fig. 7). The additional OCL constraints express the fact that both the line start and end boxes are mapped to domain elements, which serve as a source and target respectively for the domain image of the line. The constraints are the following:

Context ControlFlowLine **inv**:
start=mappedFlow.source.symbol and
end= mappedFlow.target.symbol

Here the assumption is made that *start* leads to the "geometrical" start box of the line.

The described mapping types are sufficient for defining this simplified activity diagram. But certainly our environment has more mapping types, which cover all the typical diagramming patterns (the mapping type library contains about 20 such mappings). All of them can be defined in a way similar to those presented in this paper – their syntax as mapping schemas and semantics as OCL constraints.

Diagram definition facilities in GMT

Mappings between the domain and presentation packages of the metamodel form the basis for diagram definition in our GMT approach. To define a diagram type, mappings for the diagram itself and all its elements – boxes and lines must be defined. To define a mapping in GMT environment, one has to select the appropriate mapping type from the library and to specify all the required syntax elements – appropriate metamodel classes and associations. The semantics of each mapping type is predefined in GMT (in the way presented in this paper), therefore there is no need for explicit OCL in normal cases. Certainly, for each mapping a lot of technical details may be specified – one or more default styles for diagram

elements, corresponding icons in the palette etc. For lines an important aspect is to specify, between which pairs of box types they may be drawn. Normally it is just a list of type pairs. But it is also one of the places where explicit OCL may be of use, in order to specify context dependent constraints, especially on multiplicities, present in some modeling notations. These constraints typically are defined in the presentation package. Some other diagram integrity constraints expressible in OCL are also available. A special case of line mapping is that mapped to a "pseudoline" – box nesting, available in our core and used e.g., for nested states in UML statecharts.

Yet another aspect is how the model data – attributes of domain classes and contents of subordinate classes (e.g., class operations) are mapped to graphical text slots of diagram symbols (lines) – compartments. For each compartment an OCL-style "navigation expression" points to the domain attribute, which supplies that value. For example, in the case of mapping 1OT this expression contains just the role name of the mapping link and the attribute name. More than one data supplier expression can be used for complex compartments, and navigation expressions yielding a set of values are used for "list compartments", such as class attributes or operations in UML. The way how the actual string in a complex compartment is composed from the selected data values is specified by means of a "pattern" - a simple regular grammar. Certainly, the definition component in GMT always prompts the most typical values for compartment definitions, so in simple cases nearly everything is provided automatically.

When the appropriate diagram definitions are supplied, GMT acts as a commercial modeling tool for the given notation, with all typical services enabled.

Alternative diagrammatic representations

One interesting aspect of GMT is the possibility to have different graphical representations for the same domain data. This is accomplished by defining more than one mapping (including a diagram and all of its elements) for a domain diagram, such as *Activity*. Then all of these mappings are active simultaneously, and all the representations can be used to view or edit the model data. Which views are really visible, is defined via the model browser (tree) specifications – a topic out of scope for this paper. When the model (domain) data are modified through one of the diagram views, the alternative ones have to be automatically updated also – this process is called **consolidation** in GMT. Since the mappings actually are bidirectional (due to symmetrical constraints in the scaffolding schema), the updates of diagram elements when domain elements change in general are straightforward. In some special cases OCL preconditions for consolidation may be used. But certainly all this assumes the existence of a "domain diagram" (*Activity*, *StateMachine*, *Interaction* etc), which determines what really must be inside one graphical diagram. Though some other patterns (mapping types) in GMT permit a situation such as for UML class diagrams where no "domain diagram" exists, alternative representations require such one.

To give some insight into alternative graphical representations, we will sketch briefly, how our simple activity domain could be represented via simplified ARIS eEPC [8] diagrams (see example in Fig.10). Here we assume that a (mandatory) named event symbol (hexagon) between two function (=action) symbols actually represents a named control flow (possibly with a guard). The event could be treated also as an object flow in activity notation, but then eEPC had no control flows at all and the alternative mapping would be nearly the same as that for activity diagram with object flows, having only different presentation classes. Fig. 8 shows the set of mappings between a simplified eEPC diagram presentation package and the same activity domain (with DiagramCore in the leftmost column). We remind that mapping associations are displayed bold in the picture.

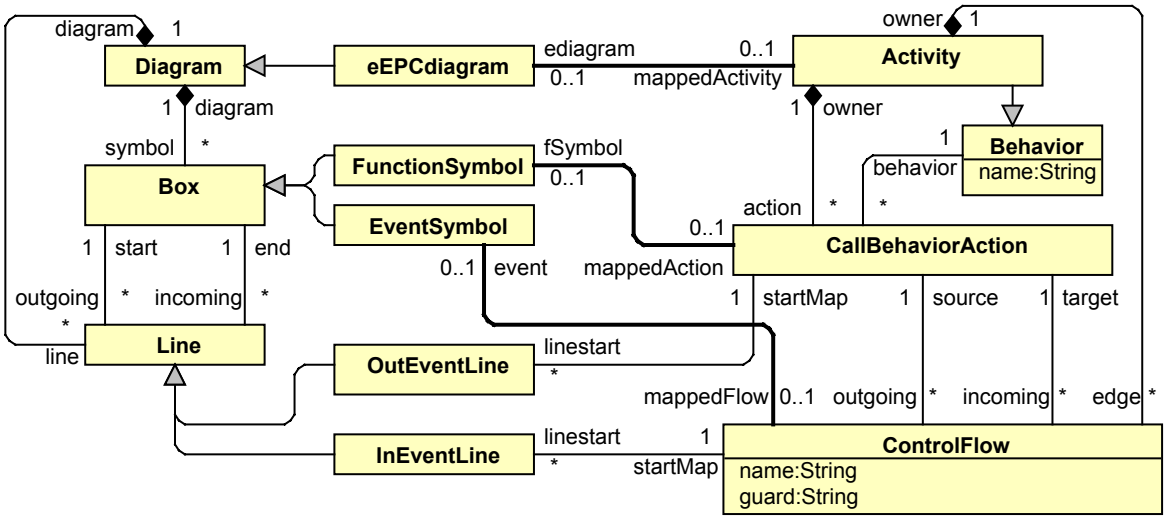

Figure 8. ARIS eEPC Mapping to activity domain

It can be seen, that the eEPC function symbol is mapped in a normal way (as for activity diagram) to *CallBehaviorAction*, using the mapping type 1OTD. The event symbol is mapped to *ControlFlow* (which was mapped to a line in activity diagram!) – the mapping type is 1OT. The attributes of *ControlFlow* – *name* and *guard* are combined into the event name compartment. Both lines types in eEPC (from function to event and vice versa) have a new mapping type L1LT – each of them actually corresponds to a domain association, but not a class. This type of mapping is based on so-called startMap – a specially built association to the domain class, where the desired association starts, with the required association role specified as a constraint (e.g., *target* from *ControlFlow* for the *inEventLine*). It should be noted that L1LT–mapped lines can have no texts – there is no place for data in the domain.

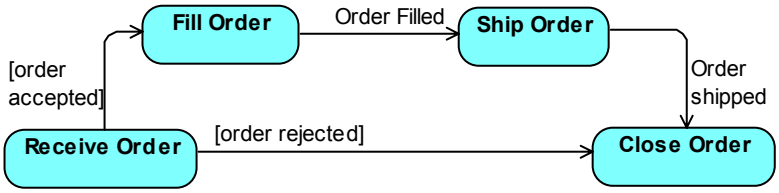

Figure 9. Activity diagram example

Thus the alternative mapping of the activity domain has been defined (certainly, the mapping details are visible only in the real GMT definition component). The consequences for GMT are the following – if we draw an activity diagram (in Fig.9), then the equivalent eEPC diagram (in Fig. 10) is drawn automatically via the consolidation process mentioned above. The activity diagram is clearly a fragment of the real notation – only the elements defined in Fig. 1 and 2 are used.

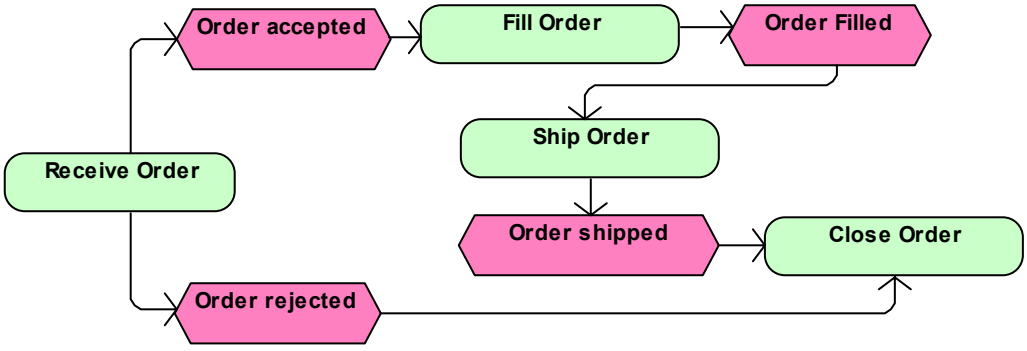

Figure 10. Equivalent ARIS eEPC diagram example

Alternative mappings for the same domain, provided in GMT, can be extended to complete UML 2.0 activity diagram (actually, its business modeling subset, full domain is much richer than that meaningful for ARIS) and complete ARIS eEPC diagram. In most cases the mappings are quite similar (the same domain concept is represented by boxes in both notations, e.g., flow join symbol and AND-rule). The non-trivial cases are when a box is used in one notation and line in another – besides the one explained in Fig. 8 a similar problem is for performer notation.

In general, when a domain for one modeling notation is found to be usable (from the semantics point of view) for another one, the building of alternative mapping can be started. It is done by drawing a candidate mapping association from a presentation class to the domain class, which most naturally corresponds to this presentation class and contains the main data to be shown in the symbol. Then, looking at adjacent domain classes, which may also contain relevant data or determine the connections, the appropriate mapping type from the library is found (there may be cases, when one presentation class maps to a structure of 2 or 3 domain classes). And conversely, two presentation classes may map to the same domain class (the "derived" mapping types are also provided). For lines, the possibility for L1LT mapping (as in Fig. 8) must also be checked, as well as "pseudoline" (box nesting) case. All this determines the supported variations between both notations – what is one symbol in one notation, can be several ones in another, a box may become a line and vice versa, but there must be some "local correspondence" anyway. Currently there is no formal procedure for deciding whether an alternative mapping can be built, it is a subject for future research. Simply, in all practical situations we have succeeded – it is more a question of the mapping library completeness. And, certainly, the main question – whether two modeling notations are semantically equivalent and in principle can have a common domain – is completely out of scope for this formal approach.

It should be noted that a price has to be paid for the universality of our approach – even in simple cases where no alternative representations of a domain are planned, two sets of classes – for domain and presentation are required by the basic technology. To avoid this excessive metamodel complexity for simple notations, a special "identity mapping" – domain and presentation classes coincide – is also provided in GMT.

Conclusions

The described method of diagram definition by mappings from presentation to domain packages is powerful enough to define the complete UML notation, including alternative presentations of interactions as UML sequence and collaboration diagrams. In addition, processes can be presented both as UML activity diagrams and traditional business process notations, such as ARIS eEPC diagrams. The approach has been tested in the GMT environment, yielding a modeling tool of industrial quality, including the efficiency for large-scale models. The alternative notations were really used for business modeling purposes, where different members of a team were provided their favorite notation. The mapping library occurred to be sufficient for diagrams of all reasonable types.

However, the approach is applicable also to a completely different area of modeling – that of MDA [9]. There series of models, typically called PIM (platform independent model) and PSM (platform specific model) are built, in order to provide a model transformation based path from requirements to system code. A typical example of a fragment of such path could be the transition from a UML class diagram for persistent data of a system (in the role of PIM) to SQL-based Data model (or ER model) in the role of PSM. Since not only the forward path is important in practice, but also the reverse one, it is reasonable to consider them as two representations of common domain data. Then classes correspond to tables, associations to relations based on foreign/primary keys and so on, each presentation showing

only the relevant aspects of the common domain. This example can completely be covered by the proposed approach and has been tested within GMT. However, some other MDA applications require true transformations of models at the domain level. We expect that the proposed metamodel mapping principles can be applied in this completely new context too.

References

[1] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P. The Generic Modeling Environment, *Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001*.

[2] DOME Users Guide, *http://www.htc.honeywell.com/dome/support.htm*

[3] MetaEdit resources, *http://www.metacase.com/papers/index.html*

[4] Kalnins A., Barzdins J., Celms E., Lace L., Opmanis M., Podnieks K., Zarins A. The First Step Towards Generic Modelling Tool, *Proceedings of Baltic DB&IS 2002, Tallinn, 2002, v. 2*, pp. 167-180.

[5] Lace L., Celms E., Kalnins A. Diagram definition facilities in a generic modeling tool, *Proceedings of International Conference Modelling and Simulation of Business systems, Vilnius, 2003,* pp. 220-224.

[6] Akehurst D. H, Kent S. A Relational Approach to Defining Transformations in a Metamodel. *In J.-M. Jezequel, H. Hussmann, S. Cook (Eds.), Lecture Notes in Computer science, Vol. 2460. Springer, 2002*, pp.243-258.

[7] Unified Modeling Language: Superstructure (version 2.0), *http://www.omg.org/docs/ptc/03-08-02.pdf*

[8] Scheer, A.-W. ARIS Business Process Modeling, *3rd edn. Springer-Verlag, Berlin Heidelberg New York (2000)*.

[9] MDA Guide Version 1.0.1, *http://www.omg.org/docs/omg/03-06-01.pdf*

[10] UML 2.0 Diagram Interchange, *http://www.omg.org/docs/ad/03-02-07.pdf*

[11] Kikusts, P., Rucevskis, P. Layout Algorithms of Graph-like Diagrams for GRADE-Windows Graphical Editors, *Lecture Notes in Computer science, Vol. 1027. Springer-Verlag, 1996*, pp.361-364.