MoDELS 2011, LNCS 6981, pp. 364-378, 2011. © Springer-Verlag Berlin Heidelberg 2011

Diagram Definition: a Case Study with the UML Class Diagram

Maged Elaasar[1,2] and Yvan Labiche[2],

[1] IBM Canada Ltd, Rational Software, Ottawa Lab
770 Palladium Dr., Kanata, ON. K2V 1C8, Canada
melaasar@ca.ibm.com
[2]Carleton University, Department of Systems and Computer Engineering
1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
labiche@sce.carleton.ca

Abstract. The abstract syntax of a graphical modeling language is typically defined with a metamodel while its concrete syntax (diagram) is informally defined with text and figures. Recently, the Object Management Group (OMG) released a beta specification, called Diagram Definition (DD), to formally define both the interchange syntax and the graphical syntax of diagrams. In this paper, we validate DD by using it to define a subset of the UML class diagram. Specifically, we define the interchange syntax with a MOF-based metamodel and the graphical syntax with a QVT mapping to a graphics metamodel. We then run an experiment where we interchange and render an example diagram. We highlight various design decisions and discuss challenges of using DD in practice. Finally, we conclude that DD is a sound approach for formally defining diagrams that is expected to facilitate the interchange and the consistent rendering of diagrams between tools.

Keywords: Diagram, Definition, Model, MOF, UML, QVT, DD, SVG

1 Introduction

Model-driven engineering (MDE) is a software methodology that is based on the use of models as a primary form of expression. Models are defined as instances of a metamodel, a higher-level model that describes the abstract syntax of a modeling language. Those languages are either general-purpose like UML [1] or domain-specific (DSML) like BPMN [2]. In fact, metamodels are themselves defined using a DSML called MOF [3]. In addition, models are interchanged between MOF-based tools in XMI [4], a specification that maps MOF to XML.

Moreover, most modeling languages (including the ones aforementioned) have a graphical concrete syntax, i.e., a diagrammatic notation. In fact, some tools (e.g. Microsoft Visio [5]) create models strictly based on the notation. Unfortunately, such a notation and its relation to the language's abstract syntax are often loosely and informally defined using text and figures (showing examples of notation). This lack of precision and formality prevents tools from interchanging modeling diagrams reliably. It also leads to inconsistent rendering of diagrams among tools, which hinders interpretation by users. This led the OMG standards body to issue a request for proposal [6] to address this problem. As a result, a new specification named Diagram Definition (DD) [7] has emerged. The specification, whose formalization is still under way [23], provides an architecture allowing the specification of (1) the diagram interchange (DI) and (2) the diagram graphics (DG) mapping for any modeling language. DI is used to define the graphical aspects that are user controllable whereas DG mapping is used to define the graphical aspects that are specified by the language (and therefore uncontrollable by the user). The role DD plays in specifying the concrete syntax of a modeling language is akin to the role MOF plays in defining the abstract syntax of that language.

In this paper, we report on a case study where we validate the DD architecture by formally specifying the Diagram Definition of a subset of the UML class diagram. First, we define an interchange syntax for this subset with a MOF-based metamodel named UML DI (that is an extension of a more generic DI metamodel provided by DD). This metamodel together with the abstract syntax metamodel represent what is needed to reliably interchange this subset between tools. Second, we define the graphical syntax of the subset by mapping the two interchange metamodels to a generic DG metamodel (provided by DD) with a QVT-based [9] transformation. The mapping rules specify how the chosen subset should be rendered to graphics.

We then carried an experiment where we exported an example class diagram from the native format of a modeling tool to the standard UML metamodel and our UML DI metamodel. We also prototyped rendering the exported diagram to graphics based on our specified DG mapping. Results showed that we could effectively interchange the example diagram and render it consistently with the original tool.

The rest of this paper is structured as follows: Section 2 provides an overview of DD and its architecture; a case study of using DD to define a subset of the UML class diagram is presented in Section 3; Section 4 describes an experiment where the definition was used to interchange and render an example diagram; a discussion and reflection on the case study are given in Section 5; Section 6 highlights related works; and finally conclusions and future works are provided in Section 7.

2 Overview of Diagram Definition

2.1 Architecture

The Diagram Definition (DD) specification [7] provides a basis for defining graphical notations, specifically node and arc style diagrams, where the notations are tied to abstract language syntaxes defined with MOF. DD provides an architecture that distinguishes two kinds of graphical information: one that users can control, such as layouts and notational options, is captured for interchange between tools; another that users do not control, such as normative shape and line styles defined by a language specification, is not interchanged because it is the same across all tools conforming to the language. DD defines two metamodels to enable the specification of these two kinds of graphical information: Diagram Interchange (DI) and Diagram Graphics (DG), respectively.

The DD architecture (Figure 2.1) resembles a typical model-view-controller architecture [8], which separates views from underlying models, and provides controllers to keep them consistent. The model part of the architecture is represented by diagram elements and their associated model elements, both of which are created by end users and thus need to be interchanged between tools. Model elements are instances of an abstract syntax (AS) metamodel; while diagram elements are instances of a related diagram interchange metamodel (AS DI). Both metamodels are defined by a language specification (e.g., UML) as instances of MOF. The AS DI metamodel is also defined as a specialization of the more abstract DI metamodel provided by DD. On the other hand, the view part of the architecture is represented by graphical elements that are defined as instances of the MOF-based DG metamodel, provided by DD to represent platform-independent graphics. Finally, the controller part of the architecture is represented by a mapping from the interchanged data (diagrams and models) to the viewable/rendered data (graphics). This mapping, which is part of a language specification, formally encodes the language's concrete syntax (CS) rules and can be expressed using any suitable mapping language (e.g., QVT [9]). The M-levels in Figure 2.1 are layers of the metamodeling architecture described in [10].

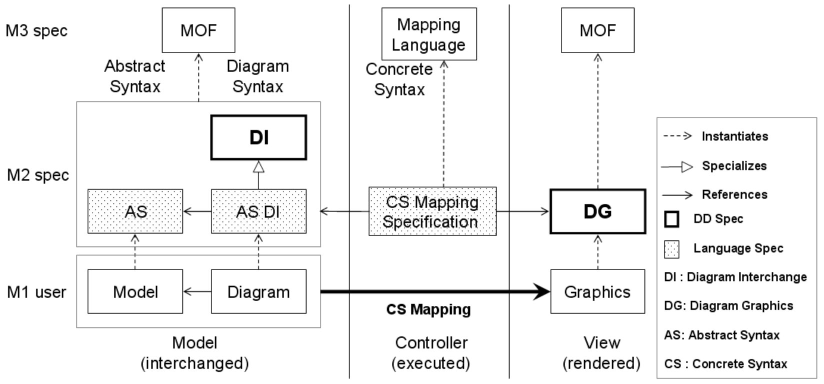

Figure 2.1 Diagram Definition Architecture

2.2 Diagram Interchange (DI)

DD provides a DI metamodel (shown in Figure 2.2 after incorporating the first official change ballot [23]) that allows defining those aspects of graphics that a language specification chooses to give its users control over and that needs to be interchanged. Rather than providing a fit-for-all metamodel, DD provides a high-level metamodel that is intended to be specialized by each language to meet its specific needs while conforming to the same best practices.

The core class in DI is *DiagramElement*, which is the super class of all elements nested recursively (via *ownedElement*) in a diagram. A diagram element can be a depiction of a *modelElement* from an abstract syntax model (e.g., a UML component) or can be purely notational (e.g., an attribute compartment). It can also inherit a style (with visual properties such as colors and fonts) from a nesting element, have a *sharedStyle* with other elements and even have its own *localStyle*. A diagram element is laid out based on being an instance of *Shape* or *Edge*. A shape is laid out within its *bounds* while an edge is laid out as a poly line with a list of *waypoint* going from a *source* element to a *target* element. Also, an element is always rendered on top of its owning element if they overlap. *Diagram* is a special kind of shape that establishes a new coordinate system for its nested elements. The top-left corner of a diagram is the origin and all location and size measurements are in device units (i.e., pixels).

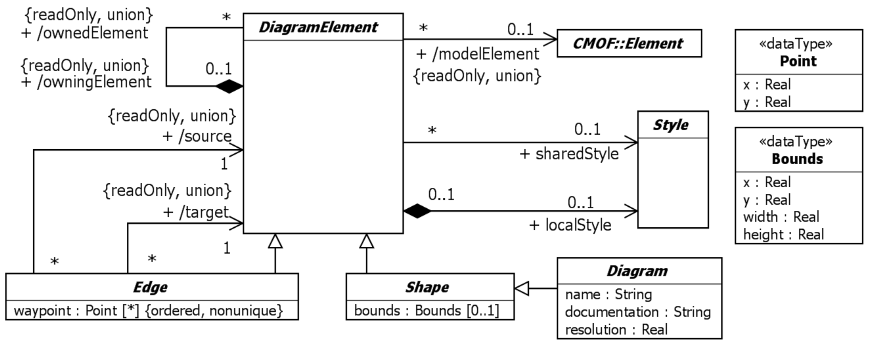

Figure 2.2 DI Metamodel

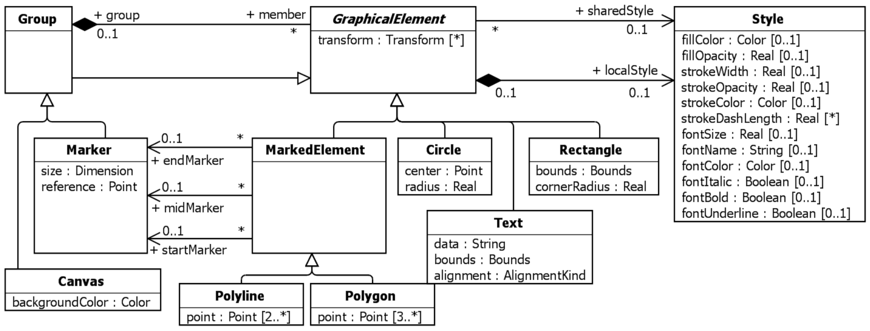

Figure 2.3 DG Metamodel (excerpt)

2.3 Diagram Graphics (DG)

DD provides a DG metamodel (Figure 2.3) that allows specifying the concrete syntax of languages in a platform-independent way with 2D graphical information. The core class in DG is *GraphicalElement*, which is the super class of all elements nested in a canvas. An element can either be a primitive (e.g., *Rectangle*, *Circle* and *Text*) or a *Group* containing *member* elements. It can inherit a style (with visual properties such as *fillColor* and *fontName*) from a group it belongs to, have a *sharedStyle* with other elements and even have its own *localStyle*. Some primitives are defined as a

MoDELS 2011, LNCS 6981, pp. 364-378, 2011. © Springer-Verlag Berlin Heidelberg 2011

connected set of points (e.g., *Polygon* and *Polyline*) and may be decorated with markers (groups of elements) at the start, middle and end points. A *Canvas* is a special kind of group used as a root of containment in a graphics model.

3 Case Study: UML Class Diagram Definition

In this section we report on a case study where we used the DD architecture to formally define the UML class diagram, both in terms of interchange and concrete syntax mapping. We choose the class diagram due to its widespread use and familiarity. However, to contain the effort, we limited ourselves to a representative subset consisting of three classifiers (*Class*, *Interface* and *DataType*) and three relations (*Association*, *Generalization* and *InterfaceRealization*). We believe this subset exemplifies the notation (shapes with labels, compartments and alternative graphics—edges with labels, markers and line styles) of the class diagram.

3.1 Diagram Interchange

Before defining the UML DI metamodel, we set some ground rules to govern our design decisions. (1) We avoid interchanging notational information that can be derived from the UML model to minimize redundancy between the DI and UML models. (2) We interchange simple layout constraints (bounds for all shapes/labels and waypoints for all edges) and avoid constraints of more complex layout algorithms to make it easier for tools to map to/from their native layouts. (3) We interchange the overlapping order of sibling diagram elements (which can happen when a diagram is crowded) by making all nested element collections ordered (a higher index implies a higher overlap order). (4) We avoid interchanging purely stylistic properties (e.g., colors/fonts) that tools may give users control over since they may vary dramatically between tools. However, we made an exception to some font properties (e.g., name and size) that we suspected could affect layout. (5) We keep the DI class hierarchy small, thus easier to maintain and evolve, by avoiding extensive sub-classing (resembling the UML class hierarchy). Instead, we allow DI classes to have a mixed bag of optional properties that apply in specific UML contexts only.

Furthermore, we defined the UML DI metamodel (Figure 2.2) by extending the DI metamodel, where appropriate, using MOF's extension semantics (subclassing and property subsetting and redefinition). Specifically, we defined class *UMLDiagram* that composed a collection of elements of type *UMLDiagramElement*. The latter could optionally reference an element from a UML model and could be styled with instances of class *UMLStyle*, which had two properties (*fontName* and *fontSize*).

Then, we defined classes for interchanging the chosen shapes and edges of the class diagram. To do that, we analyzed the relevant notation in the UML specification and identified three cases (shown in Figure 3.2): (a) a shape that has a label and an optional list of compartments, each of which having an optional list of other labels (e.g., the classifier box notation); (b) a shape that has a label only (e.g., the interface ball notation); and (c) an edge that has an optional list of labels (e.g., the association notation). However, (b) is really a special case of (a) when there is no compartment.

Based on that, we defined three shape classes (*UMLShape*, *UMLLabel* and *UMLCompartment*) and one edge class (*UMLEdge*) and related them with the multiplicities in cases (a) and (c). We also defined them (except *UMLCompartment*) as subclasses of *UMLDiagramElement* to allow them to be styled separately, reference their own UML elements, and be connectable (an edge was made to only connect elements of that type). We then added some properties to disambiguate the notation. For example, a *kind* can be set on a label to indicate what aspects of the UML element to show textually. A flag *showClassifierShape* can be set on a classifier's shape to indicate whether to use the *box* notation. Notice that we only added a subset of the possible notational options for brevity.

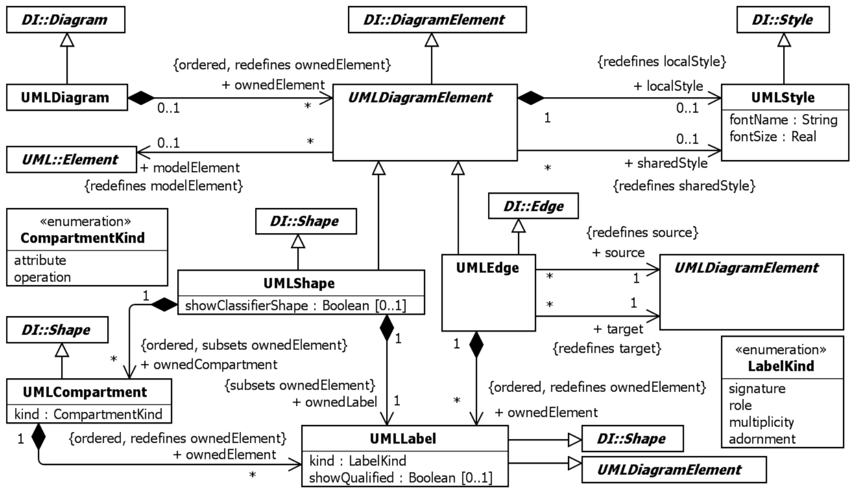

Figure 3.1 UML DI Metamodel

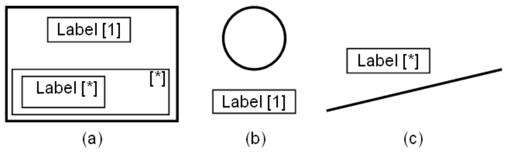

Figure 3.2 Class Diagram Notational Patterns

3.2 Concrete Syntax Mapping

Recall from Section 2.1, that a language can specify its concrete syntax rules as a mapping from the AS DI metamodel, which references the AS metamodel, to the DG metamodel. The mapping can be expressed using any mapping language. We chose to express it using the QVT Operational (QVTo) [9] transformation language (designed as a thin extension of OCL [11]). The choice was motivated by our previous experience with QVTo, the fact that it is a standard MOF-based language, the fact that it is executable (allowing us to test our mapping), and the availability of a good implementation [12]. Due to space limitation, we only show parts of the transformation we defined. We assume some reader's familiarity with QVTo and OCL. We also assume familiarity with the syntax of the UML metamodel [1].

The class diagram's concrete syntax is defined with a QVTo transformation from a UML DI model to a DG model (Figure 3.3, line 1). The transformation starts by looking for all instances of *UMLDiagram* and initiating the mapping for them (lines 2-4). Mappings (like operations) are defined on UML DI classes and have DG classes as return types. For example, a mapping named *toGraphics* is defined on the *UMLDI::UMLDiagram* and has *DG::Canvas* as a return type (line 5). This maps an instance of *UMLDiagram* to an instance of *Canvas*, and initializes the properties of the latter according to the body of the mapping. In this case, the body iterates on all the owned elements of the diagram, mapping each one in turn to graphics, and adding the resulting graphical elements as members of the canvas (line 6).

```
01  transformation UMLDIToDG(in umldi : UMLDI, out DG);
02  main() {
03     umldi.objectsOfType(UMLDiagram)->map toGraphics();
04  }
05  mapping UMLDiagram::toGraphics() : Canvas {
06     member += self.ownedElement->map toGraphics();
07  }
08  mapping UMLDiagramElement::toGraphics() : Group {
09     localStyle := copyStyle(self.localStyle);
10     sharedStyle := copyStyle(self.sharedStyle);
11  }
12  mapping UMLShape::toGraphics() : Group
13     inherits UMLDiagramElement::toGraphics {
14     member += self.modelElement.map toGraphics(self);
15     member += self.ownedLabel.map toGraphics ();
16     member += self.ownedCompartment->map toGraphics();
17  }
18  mapping UMLEdge::toGraphics() : Group
19     inherits UMLDiagramElement::toGraphics {
20     member += self.modelElement.map toGraphics(self);
21     member += self.ownedElement->map toGraphics ();
22  }
23  mapping UMLCompartment::toGraphics() : Group {
24     member += object Rectangle {bounds := self.bounds};
25     member += self.ownedElement->map toGraphics ();
26  }
27  mapping UMLLabel::toGraphics () : Text
28     inherits UMLDiagramElement::toGraphics {
29     var e := self.modelElement;
30     var q := self.showQualified;
31     bounds := self.bounds;
32     data := switch {
33        case (self.kind = LabelKind::signature)
34           e.oclAsType(NamedElement).getSinature(q);
35        case (self.kind = LabelKind::role)
36           e.oclAsType(Property).getRole();
37        ...
38     };
39     localStyle := e.map toStyle(self); // update style
40  }
```

Figure 3.3 QVTo Mapping from Class DI to DG

Furthermore, a *UMLShape* maps to a *Group* (lines 12-17) consisting of the following: a graphic for the model element (line 14), a graphic for the owned label (line 15) and a graphic for each owned compartment (line 16). These graphics are produced by other nested mappings (shown later). A similar mapping is defined for *UMLEdge* (lines 18-22). However, the mapping for *UMLCompartment* (lines 23-26) is different as the first member graphic is fixed as a *Rectangle* whose bounds are defined by the compartment. The mapping for *UMLLabel* (lines 27-40) is also different as it maps to a *Text* whose bounds are defined by the label and whose data value is defined based on the label kind. For example, if the kind is *signature*, the value is defined by a query *getSignature* defined on *NamedElement* (line 33-34). Also notice how the mapping inherits (line 28) another mapping (lines 8-11) that copies over the local and shared styles. The local style is further updated (line 39) based on the label's model element (e.g., the *fontItalic* property is set to *true* for the *signature* label in the case of an abstract classifier).

Some of the queries used for the label mapping are shown in Figure 3.4. The *getSignature* query (lines 1-3) returns the (simple or qualified) name of an element based on a flag. The query is overridden for different UML types to specify their unique signatures. For example, *Interface* (lines 4-6) overrides it to prefix the name with the «Interface» keyword. *Property* (lines 12-17) overrides it to return the full signature of a property in an attribute compartment (with type, multiplicity, etc.).

```
01  query NamedElement::getSignature(q : Boolean) : String {
02     return self.getName(q);
03  }
04  query Interface::getSignature(q : Boolean) : String {
05     return “«Interface»\n” + self.getName(q);
06  }
07  query Property::getSignature(q : Boolean) : String {
08     var t := if self.type->notEmpty() then ":" +
09             self.type.getSignature(q) else "" endif;
10     return self.getRole()+ t + self.getAdornment();
11  }
12  query Property::getRole() : String {
13     var d := if self.isDerived then “/” else “” endif;
14     var v := if self.visibility = VisibilityKind::public
15        then “+” else ... endif;
16     return d + v + self.getName(false);
17  }
18  query NamedElement::getName(q : Boolean) : String {
19     return if q then self.qualifiedName
20            else self.name endif;
21  }
22  query Property::getAdornment() : String {
23     return “{“ + ... + “}”;
24  }
```

Figure 3.4 Queries Used by the UML Label Mapping

Figure 3.5 shows mappings between UML classifiers and their corresponding graphical elements (e.g., box or ball notation). The first mapping (lines 1-3), defined on UML *Element*, delegates to other mappings depending on the type of the element. Notice that both *Class* (lines 4-6) and *DataType* (lines 7-9) have one mapping each creating a rectangle, while *Interface* has two mappings, one creating a rectangle (lines 10-13) and the other creating a circle (lines 14-20), based on the flag *showClassifierShape* (lines 11, 15) on *UMLShape*.

```
01  mapping Element::toGraphics(s:UMLShape):GraphicalElement
02     disjuncts Interface::toRectangle, Interface::toCircle,
03               Class::toRectangle, DataType:toRectangle {}
04  mapping Class::toRectangle (s:UMLShape) : Rectangle {
05     bounds := s.bounds;
06  }
07  mapping DataType::toRectangle (s:UMLShape) : Rectangle {
08     bounds := s.bounds;
09  }
10  mapping Interface::toRectangle (s:UMLShape) : Rectangle
11     when { s.showClassifierShape=true } {
12     bounds := s.bounds;
13  }
14  mapping Interface::toCircle (s:UMLShape) : Circle
15     when { s.showClassifierShape=false } {
16     var b := s.bounds;
17     center := object Point{b.x+b.width/2;b.y+b.height/2};
18     radius := if b.width<b.height then b.width/2
19               else b.height/2 endif;
20  }
```

Figure 3.5 UML Classifier Mappings to Graphics

Figure 3.6 shows mappings between UML relations and poly lines. The first mapping (lines 10-13), defined on UML *Element*, delegates to other mappings depending on the type of the element. The mapping of relation *InterfaceRealization* (lines 14-19) copies the edge's waypoints to the poly line'a points (line 15). As the notation of this relation depends on whether the interface shape was shown as a box or a ball, this is checked first (line 16). If it is shown as a box, a shared style with a dash pattern (lines 1-2) and a closed arrow marker (lines 3-9) are used (lines 17-18).

```
01  Property interfaceRealStyle = object DG::Style {
02     strokeDashLength := Sequence {2, 2} };
03  property interfaceRealMarker = object Marker {
04     size := object Dimension {width := 10; height := 10};
05     reference := object Point {x := 10; y := 5};
06     member += object Polylgon {
07        point += object Point{ x:=0; y:=0 };
08        point += object Point{ x:=10; y:=5 };
09        point += object Point{ x:=0; y:=10 }; }; };
10  mapping Element::toGraphics(e:UMLEdge):GraphicalElement
11     disjuncts Association::toPolyline,
12               Generalization::toPolyline,
13               InterfaceRealization::toPolyline {}
14  mapping InterfaceRealization::toPolyline(e:UMLEdge):Polyline{
15     point := e.waypoint;
16     var s = e.target.showClassifierShape;
17     sharedStyle := if s then interfaceRealStyle endif;
18     endMarker := if s then interfaceRealMarker endif;
19  }
```

Figure 3.6 UML Relations Mappings to Graphics

4 Experiment: Interchange and Rendering a Diagram

In this section, we report on an experiment where we used the UML DI metamodel (Section 3.1) and the concrete syntax mapping (Section 3.2) we defined for the UML class diagram to interchange and render an example class diagram (Figure 4.1). The diagram was created using the Rational Software Architect (RSA) v8.0 modeling tool [13] and included model elements for the notational subset we defined. The objective of the experiment was to test the newly defined UML DD architecture in terms of its ability to interchange and consistently render diagrams between tools.

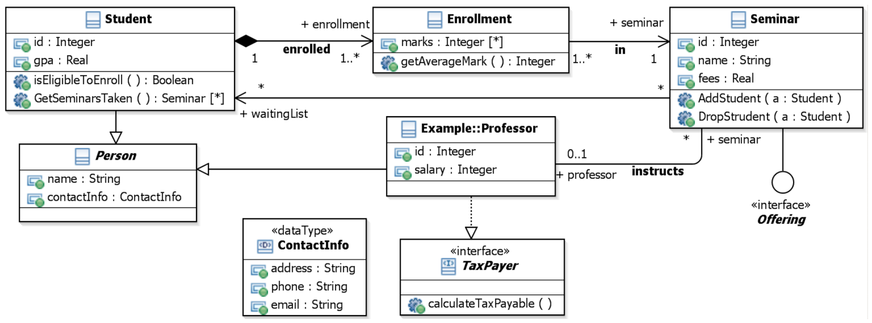

Figure 4.1 Example Class Diagram Defined in RSA

4.1 Experiment Setup

In order to satisfy our objective, we needed to export the example diagram from tool “A” in UML DI, import it into tool “B” and visually compare the diagram rendered in both tools. Since RSA played the role of tool “A” in this execution chain, we implemented a UML DI exporter for RSA. Moreover, instead of implementing an importer for another UML CASE tool “B”, we decided to implement a simple UML DI visualization tool (Section 4.2), which leveraged the UML DI to DG mapping we had specified. This allowed us to test both the effectiveness of the UML DI metamodel and the accuracy of the mapping in the same time. Additionally, such a tool can be used by other UML CASE tools to verify their own UML DI exporters.

We used the open-source Eclipse Modeling Framework (EMF) [14] project, which is packaged and used by RSA, as our MOF-based modeling tool infrastructure. We used EMF to import the two standard metamodels (DI and DG) provided by DD. We also used EMF to define our UML DI metamodel as an extension of DI. Moreover, RSA comes packaged with the open-source M2M/QVTo [12] project, which provides a QVTo editor and execution environment. We used this project to author and execute our concrete syntax mapping between UML DI and DG.

4.2 Experiment Execution

The first step of the experiment was to export the diagram into UML DI. To do that, we defined an exporter from RSA’s native diagram format into UML DI. RSA’s native format is based on a notation metamodel provided by the Graphical Editing Framework (GMF) [15]. GMF’s metamodel is in fact close in many aspects to the standard DI metamodel, so we implemented an exporter as a QVTo transformation between the two metamodels. However, we did not find all the needed layout data represented in the example GMF diagram (the bounds of some labels and shapes are derived). We worked around that by doing some pre-processing of the diagram (we rendered it using RSA, obtained the missing layout data from graphics and added them as annotations to the diagram). Support for DD in RSA should facilitate this procedure in future versions of the tool.

The second step was to render the exported diagram in another tool and visually compare it with the original RSA diagram. Our strategy for implementing such a tool consisted of two steps: 1) executing the concrete syntax mapping from UML DI to DG to obtain a resulting DG model (fortunately, since our mapping was done with QVTo, we simply executed the transformation to get a DG model); 2) rendering the DG model to graphics. For that step, we defined a model-to-text transformation to map the (used subset of the) DG metamodel to SVG [17] (DG's design is close to SVG's). We used the JET framework [25] that is packaged with RSA to define the model-to-text transformation. We then used a web browser to view the resulting SVG image. The experiment's complete execution chain is depicted in Figure 4.2.

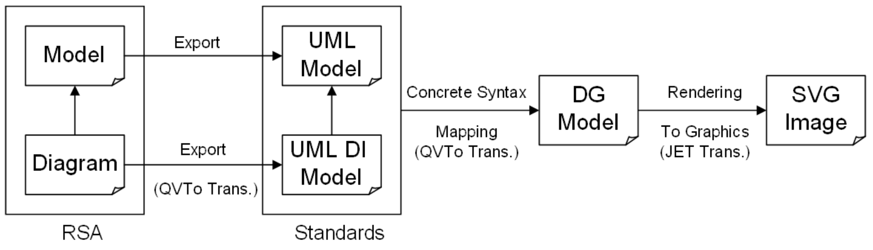

Figure 4.2 The Execution Chain of the Experiment

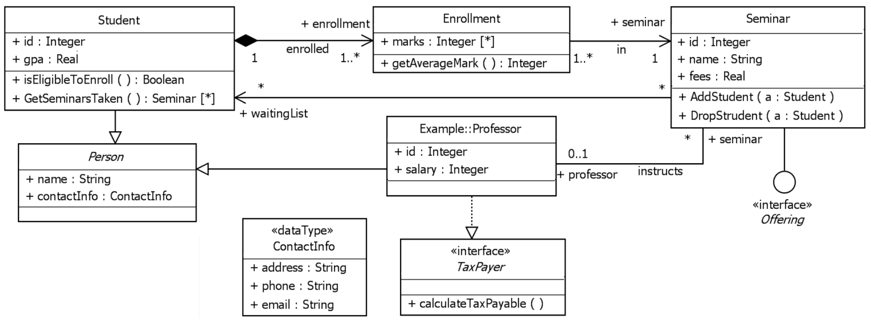

Figure 4.3 Example Class Diagram Exported and Rendered as SVG

4.3 Experiment Result

We executed the experiment on the example diagram and the resulting SVG image is shown in Figure 4.3. This image resembles to a large extent the original diagram in RSA, indicating an overall successful interchange. However, the two diagrams were not identical due to RSA’s own variations on the original notation (e.g., shapes have drop shadows, bold names, metaclass icons and list item visibility icons—edges have round edges at the corners). These variations caused some small differences in the diagram layout. Obviously, bugs in the last two links of the execution chain (Figure 4.2) could have also caused the diagrams to differ. In order for this chain to be an effective way for tool vendors, or an interchange testing group similar to [16], to test their diagram exporters, the last two links must be carefully tested first. This would make vendors focus on testing their exporters only. Obviously, this chain does not test each tool’s own importer, which needs to be tested separately.

5 Discussion

The case study and experiment show that DD is a promising approach for formally defining diagrams of MOF-based languages. By defining a language-specific DI metamodel, tools of that language can precisely interchange modeling diagrams. Also, by formally specifying a mapping from language-specific DI and AS models to DG, vendors can build tools more accurately and with less cost. Users can also reliably interpret diagrams produced by different tools.

Nevertheless, there are a number of current limitations that need to be addressed before DD is finalized. One of those limitations is a need for a normative mapping from DG, which is basically a platform-independent graphics metamodel, to at least one standard vector-based graphics format (e.g., SVG [17]). This would help the testing process as discussed earlier, but it would also help bootstrap the DD architecture by providing a concrete syntax mapping for DG (without requiring the use of DD for that). Another limitation exists when a graphical syntax is specified using a textual mapping language (like QVTo). While it is very formal and flexible, it is also less readable (compared to the current way of defining diagrams that is more readable but less formal). One alternative could be to use a mapping language that has a graphical notation like QVT Relations (QVTr) [9]. Another alternative is to define a DD-specific graphical mapping language (e.g., a BNF grammar that incorporates graphical symbols). Such language can make a mapping more readable while still being formal. Moreover, another limitation with DD is the lack of reusable standard libraries to jumpstart a new DD-based specification. One library could provide a set of pre-defined DG types (e.g., styles and markers) that are commonly used in modeling notations. Another could be a general-purpose DI metamodel for annotations (e.g., notes and their attachments) that can integrate with any language-specific DI metamodel. Such metamodel would have its own pre-defined mapping to DG. Finally, it would also help if the DD specification highlighted common design options and best practices for DD users to benefit from.

In fact, one of the most important design decisions when defining a language-specific DI metamodel is how far you go in using the MOF extension mechanisms to precisely define the DI syntax. One extreme is to go all the way such that every AS class maps to a unique DI class with applicable properties. This option makes it easier to create valid DI models (as the metamodel becomes very restrictive) but harder to maintain the metamodel (as it becomes very sensitive to changes in the AS

MoDELS 2011, LNCS 6981, pp. 364-378, 2011. © Springer-Verlag Berlin Heidelberg 2011

metamodel). The other extreme is to settle with a very small hierarchy of DI classes with properties applicable to many AS classes and provide constraints for their applicability. With this option, it becomes more difficult to create valid DI models (since constraints are checked only after creation) but the metamodels become easy to maintain and less sensitive to changes in the AS metamodels. A more pragmatic option is always somewhere in between these two extremes.

Another subtle but interesting point with using QVTo to express a concrete syntax mapping is the fact that QVTo is a unidirectional language. Therefore, what is expressed is how the abstract and diagram syntaxes map to graphics, but not the other way around. While a unidirectional mapping will still help a user interpret a diagram (i.e., relate it back to the AS syntax), it may not always work especially when the notation is ambiguous. In this case, a bidirectional mapping (e.g., with QVTr) is more preferable. However, we believe that removing ambiguity from a graphical notation (if possible) goes further than trying to address it with a bidirectional mapping.

6 Related Works

Two categories of works are discussed here: those related to diagram interchange and those related to concrete syntax mapping. One early work in the first category is the DI v1.0 specification [18], which has been deprecated by the new DD specification. One issue with that specification is that, unlike DD, it provides a fixed interchange metamodel that is not meant for extension. This forces language-specific syntax rules (called nesting rules) and constraints to be provided informally. It also forces language-specific properties to be added through a key-value string map.

Another relevant work in this category is the notation metamodel provided by GMF [15], which is used by a number of tools including RSA and Papyrus [19]. This metamodel is similar to the one discussed above in that it is not meant for extension for a given language (although its diagram elements can have multiple styles and thus new style classes can be defined). Additionally, the diagram syntax is defined by language-specific creation factories (implemented in java). Once created, there is no metadata to help generically interpret or validate the syntax of a given diagram.

Another related work is the BPMN 2.0 specification [2], which uses (an alpha version of) DD to define a BPMN-specific DI metamodel. The metamodel is designed with minimum extension to the higher level DI metamodel. In other words, it has a small number of DI classes with properties applicable to many BPMN classes. The metamodel also represents a departure from XPDL [20], a format that has historically been used to interchange BPMN diagrams. Unlike DD, XPDL uses one schema for both AS and DI data (i.e., does not separate model from notation). Moreover, BPMN specification does not specify a concrete syntax mapping from BPMN DI to DG.

Related works in the second category also exist. The first one is GMF [15], which provides two models to map diagrams to graphics. The first one is called a Graphical Definition model, where one defines graphical elements (called figures) and associate them with notational patterns (called canvas elements) like: nodes, connections, labels and compartments. Unlike DD, these notational patterns are predefined; hence one is restricted to specify a language’s notation using them only, which is inflexible. The second one is a Mapping model, where one defines mappings from AS classes to notational patterns (defined in the first model). Mappings are defined in a containment hierarchy starting from a canvas mapping, down to node and connections mappings, then label and compartment mapping. The mapping details are expressed with OCL. This strict containment hierarchy prevents mappings from being reused in other places in the hierarchy. On the other hand, DD allows mapping rules to be reused by flexibly calling them from other rules.

Another related work in this category is contributed by Palies [21], where a transformation is defined using ATL [22] (a non-standard declarative language) between the old DI metamodel [18] and a graphics metamodel (resembling SVG). The DI metamodel in this case is used to interchange UML class diagrams even though, as mentioned earlier, it is not UML-specific. Hence, the author had to make some assumptions regarding the correct DI syntax for UML. In contrast, we captured the UML DI syntax formally with a metamodel.

7 Conclusion and Future Work

Formal diagram definition has been missing in the MOF-based modeling architecture for many years. The OMG recently released a new specification called DD with an architecture that allows for formally defining diagrams of graphical modeling languages. DD allows a modeling language to define an interchange metamodel for its diagrams and precisely map the diagrams' concrete syntax to graphics. In this paper, we verified DD by using it to formally define a subset of the UML class diagram. Specifically, we extended the DI metamodel (provided by DD) to define a UML DI metamodel used for interchanging this subset between tools. We also defined the concrete syntax of the subset by mapping its UML DI metamodel to DG (a graphics metamodel provided by DD) using a QVTo transformation. We then carried an experiment where we used those definitions to interchange an example class diagram. We designed a testing chain where a diagram is exported from a modeling tool to UML DI, transformed to DG and then rendered to SVG. The exported diagram resembled to a large extent the original example diagram indicating a successful interchange. We also highlighted a number of outstanding issues with DD including a need for a normative mapping to a standard graphics format (e.g., SVG), a need for a more readable mapping to DG and a need for some standard libraries to jumpstart a new DD specification.

Going forward, we plan to use DD to define a bigger and more complex subset of the UML metamodel, especially the sequence diagram. We also plan to use it to specify the notation of a UML profile (e.g., SysML [24]) as an extension to that of UML. Other possibilities include defining the concrete syntax of UML with a bidirectional mapping (e.g., with QVTr) to ease the interpretation of diagrams, defining a graphical mapping language specific for DD and investigating other ways to jumpstart the DD definition for modeling languages.

MoDELS 2011, LNCS 6981, pp. 364-378, 2011. © Springer-Verlag Berlin Heidelberg 2011

References

1. Unified Modeling Language (UML), Superstructure v2.4. ptc/2010-11-14.
2. Business Process Model and Notation (BPMN) v2.0, dtc/2010-06-05.
3. Meta Object Facility (MOF) Core v2.4. OMG ptc/2010-12-08.
4. MOF 2 XMI Mapping v2.4. OMG ptc/2010-12-06.
5. Microsoft Visio 2010. http://office.microsoft.com/en-ca/visio/
6. Diagram Definition Request for Proposal. ad/2007-09-02.
7. Diagram Definition v1.0 FTF Beta 1. ptc/2010-12-18.
8. Booch, G.: Handbook of Software Architecture. http://handbookofsoftwarearchitecture.com
9. Query/View/Transformation (QVT) v1.0. OMG formal/2008-04-03.
10. Unified Modeling Language (UML), Infrastructure v2.4. OMG ptc/2010-11-03.
11. Object Constraint Language (OCL) v2.2. OMG formal/2010-02-01.
12. Dvorak, R.: Model Transformation with Operational QVT – M2M component. http://www.eclipse.org/m2m/qvto/doc/M2M-QVTO.pdf
13. Rational Software Architect v8.0 (RSA) Open Beta. https://www14.software.ibm.com/iwm/web/cc/earlyprograms/rational/rsaob/index.shtml
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. 2nd edition, 2009.
15. Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf/
16. Model Interchange Wiki. http://www.omgwiki.org/model-interchange/doku.php
17. Scalable Vector Graphics (SVG) 1.1. http://www.w3.org/TR/SVG/
18. Diagram Interchange v1.0. formal/06-04-04
19. MDT Papyrus. http://www.eclipse.org/modeling/mdt/papyrus/
20. XML Process Definition Language (XPDL). http://www.wfmc.org/xpdl.html
21. Palies, J.: ATL Transformation Example: UMLDI to SVG. 2005. http://www.eclipse.org/m2m/atl/atlTransformations/UMLDI2SVG/UMLDI2SVG[0.04].pdf
22. Atlas Transformation Language (ATL) http://wiki.eclipse.org/M2M/Atlas_Transformation_Language_(ATL)
23. DD FTF Wiki. http://www.omgwiki.org/dd/doku.php?id=start
24. Systems Modeling Language (SysML), v1.2. formal/2010-06-02.
25. Java Emitter Templates (JET). http://www.eclipse.org/modeling/m2t/?project=jet#jet