Softw Syst Model (2018) 17:1079–1103
https://doi.org/10.1007/s10270-016-0537-x

REGULAR PAPER

Reusing metamodels and notation with Diagram Definition

Conrad Bock[1] · **Maged Elaasar**[2]

Received: 12 October 2014 / Revised: 12 April 2016 / Accepted: 31 May 2016 / Published online: 28 June 2016

Abstract It is increasingly common for language specifications to describe visual forms (concrete syntax) separately from underlying concepts (abstract syntax). This is typically to enable interchange of visual information between graphical modeling tools, such as positions of nodes and routings of lines. Often overlooked is that separation of visual forms and abstract concepts enables languages to define multiple visual forms for the same underlying concepts and for the same visual form to be used for similar underlying concepts in different languages (many-to-many relationships between concrete and abstract syntax). Visual forms can be adapted to communities using different notations for the same concepts and can be used to integrate communities using the same notation for similar concepts. Models of concrete syntax have been available for some time, but are rarely used to capture these many-to-many relationships with abstract syntax. This paper shows how to model these relationships using concrete graphical syntax expressed in the Diagram Definition standard, examining cases drawn from the Unified Modeling Language and the Business Process Model and Notation. This gives definers of graphical languages a way to specify visual forms for multiple communities.

Keywords Notation · Metamodel · Diagram Definition · Syntax

1 Introduction

Separate modeling of language visualization (*concrete syntax*) from terminology and concepts (*abstract syntax*) began at least with textual computer languages, which include visual aspects such as punctuation and reserved words in their specifications, but remove them during implementation to simplify analysis and translation to machine languages [1]. The advent of model-based approaches brought abstract syntax into language specifications, going beyond trees to graphs of language elements (*metamodels*) [2]. Models of concrete syntax followed those of abstract syntax, initially outside of language specifications, then recently as part of them. In particular, the Object Management Group (OMG) provides a standard model for concrete graphical syntax (Diagram Definition, DD), as developed by the authors [3,4], and uses it in some graphical language specifications (see Sect. 2 for more background on concrete and abstract syntax).

Separate modeling of concrete and abstract syntax is typically used to interchange computer-interpretable information about these forms between graphical modeling tools, such as positions of nodes and routings of lines. This enables graphical models to retain their appearance as they are transferred

Communicated by Prof. Thomas Kühne.

The authors thank Anantha Narayanan, David Wagner, and Denis Gagne for their helpful comments. Identification of any commercial equipment and materials is only to adequately specify procedures. It is not intended to imply recommendation or endorsement by the U.S. National Institute of Standards and Technology, nor does it imply that the materials or equipment are necessarily the best available for the purpose.This research was carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the U.S. National Aeronautics and Space Administration.

✉ Conrad Bock
conrad.bock@nist.gov

Maged Elaasar
maged.e.elaasar@jpl.nasa.gov

[1] U.S. National Institute of Standards and Technology, 100 Bureau Drive, Stop 8260, Gaithersburg, MD 20899-8260, USA

[2] Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109-8001, USA

between tools from different vendors. For example, OMG applied DD to visual forms in the Unified Modeling Language (UML), filling a long-standing gap in UML's support for interchange of diagrammatic information [5]. Before that, the second version of Business Process Model and Notation (BPMN) defined a precursor to DD specifically for reproducing the modeler-controlled visual characteristics of its diagrams across tools [6].

Often overlooked is that separate modeling of concrete and abstract syntax enables many-to-many relationships between them, helping address two important challenges in graphical language design:

1. Accommodating different visual appearances for the same language concepts. Significantly different notations for the same concepts can prevail across user communities, such as changes in graphical shapes and layout, reversal of nodes and lines, or use of tables and text instead of graphics. For example, UML provides multiple notations for interactions between objects, each designed for particular modeling needs. Languages supporting a variety of notations for the same concepts improve communication between communities using the notations and enable the same automated tools to operate on the concepts without requiring adaptation to multiple notations.
2. Reusing the same visual appearance for similar concepts in multiple languages. This is useful when multiple languages about the same topics are developed in parallel for different communities, such as the many languages for specifying processes, leading some communities to borrow notations from others. For example, UML and BPMN overlap significantly in process modeling capabilities, and some applications benefit from using BPMN to notate UML concepts [7]. To maintain efficient communication and automation within the borrowing community, it is helpful to support the foreign notation as a visualization of the community's prevalent language concepts.

Prior work with separate models of concrete and abstract syntax does not address many-to-many relationships between them for graphical languages. Sometimes the same abstract syntax is used with a single human-readable textual concrete syntax alongside a single graphical syntax [8,9], or to provide multiple textual syntaxes [10], but not with multiple graphical syntaxes as in this paper. The authors are not aware of work examining a single concrete syntax notating multiple abstract syntaxes. The flexible relationship of concrete and abstract syntax is mentioned briefly in the context of introducing complex mappings between them [11], unifying them to reduce redundancy [12,13] extracting concrete from abstract syntax [14], and upgrading textual parsers to use abstract syntax [15–17], but these do not directly address many-to-many relationships between abstract and concrete syntax as in this paper (see Sect. 2 for more about related work).

This paper demonstrates many-to-many relationships between graphical notation and language concepts, giving definers of graphical languages a way to specify visual forms for multiple communities, as needed to address the challenges above. It shows how to model multiple graphical concrete syntaxes on a single abstract syntax in cases drawn from UML and a single graphical concrete syntax notating multiple abstract syntaxes in cases drawn from BPMN, using DD for graphical concrete syntax. Section 2 reviews the history and benefits of separating abstract and concrete syntax and compares the contributions of this paper to related work on the subject. Section 3 shows how graphical concrete syntax is modeled in DD. Section 4 demonstrates modeling of multiple graphical notations on a single metamodel, and Sect. 5 does the same for a single graphical notation on multiple metamodels. Section 6 concludes the paper.

2 Background and related work on abstract and concrete syntax

Language syntax is a set of rules for constructing the physical forms of communication or specification, typically visual, and in particular, textual ("grammar") [18]. These rules define the kinds of elements in a language and how they can be combined. For example, English provides nouns, verbs, and so on that can be combined in particular orders, such as a verb appearing between two nouns. Languages for specifying system behavior have more restrictive rules, with some words predefined (*reserved*). For example, in JavaScript the predefined word "function" can appear before a word not predefined, to specify a procedure [19]. A common way to specify textual syntax is Backus–Naur Form (BNF), which is a textual language for giving rules about composing smaller pieces of text into larger ones [20,21].

In computer languages, some aspects of language syntax are not needed when translating for machine execution, such as punctuation marks, and can be simplified by removing them, producing various kinds of abstract syntaxes. These translators use object representations for pieces of text, linking objects for smaller pieces to the larger ones containing them, forming data structures (abstract syntax trees, or "parse trees") [1,22]. For example, semicolons ending statements in JavaScript can be removed before automated processing, because abstract syntax elements representing blocks of statements are linked to elements representing each statement in them. This separates and orders constituent elements without punctuation or textual proximity and makes abstract

relationships and elements explicit, such as composition, to simplify translation of the language for machine execution.

Rules for the visual form of a language are its concrete syntax, whereas rules for its computational form are abstract syntax. Rules for concrete and abstract syntax must be consistent, even though abstract syntax omits some aspects of the visual form, and add rules for abstract elements and their relationships, as described above. For example, concrete syntax for English does not allow one verb right after another, and neither does its abstract syntax. In the concrete syntax for JavaScript, the predefined word "function" must be followed by a word not predefined and then a parenthetic expression for parameters, while the abstract syntax requires the element for function definition to include an element for parameter definition, but omits the parentheses and predefined words like "function."

The advent of model-based approaches brought abstract syntax into language specifications, going beyond trees to graphs of language elements (*metamodels*).[1] One of the first was the Computer-aided software engineering Data Interchange Format (CDIF), which used metamodels to specify file formats (textual concrete syntax) for interchange of information models, including entity-relationship models and state machines [25].[2] OMG adapted CDIF's approach to specify files formatted in the eXtensible Markup Language (XML) [27], for interchange of metamodels defined in a subset of UML (Meta-Object Facility, MOF) [28], with mappings from metamodels to XML specified by XML Metadata Interchange (XMI) [29]. OMG eventually required all its metamodels, including UML, to be defined in the MOF subset of UML, with interchange file formats specified in XMI.

Models of concrete graphical syntax followed those of abstract syntax, initially outside of language specifications, then recently as part of them. The Eclipse Foundation provides open-source graphical syntax modeling for use with its UML and other modeling tools, because OMG had not addressed this when UML was originally standardized [30,31]. OMG attempted to address graphical syntax modeling in a standard accompanying UML's major revision, but the resulting metamodel for graphical syntax could only be extended for particular languages in free text, rather than by class specialization with language-specific properties, and was too closely tied to UML [32].[3] OMG's major revision of BPMN provides models of its graphical syntax, but they are not generic enough to use with other languages [6]. OMG recently addressed these problems with a new way to specify diagrams in its graphical language standards (DD, see Sect. 1), which is used in a recent version of UML and other modeling standards [5,33,34].

Despite the history and benefits of separately modeling abstract and concrete syntax described above, it has not been used to address many-to-many relationships between graphical concrete and abstract syntax, as needed to address the challenges identified in Sect. 1:

1. Multiple concrete graphical syntaxes for a single abstract syntax, enabling the same language to be adapted to multiple communities or purposes, while sharing the same automated tools. Prior work only uses the same abstract syntax with a single graphical syntax alongside multiple textual concrete syntaxes (human readable and/or tool interchangeable). For example, UML and its extensions have graphical and textual forms, where the textual form is for interchanging models between tools [5,33]. An executable subset of UML has a human-readable textual syntax that is mostly the same as a subset of Java syntax, giving that portion of UML both graphical and textual concrete syntax on the same abstract syntax [8,35,36]. BPMN has a graphical syntax for modelers and two textual ones for model interchange, one defined in XML Schema Definition Language (XML Schema) and the other in XMI [6,29,37]. The Ontology Web Language has multiple textual notations based on one abstract syntax ("structural specification"), though this is only specified in diagrammatic form, without interchange files [10].[4]
2. Single graphical concrete syntax for multiple abstract syntaxes, enabling the same notation to be used in the context of multiple languages. The authors are not aware of prior work examining this aspect of many-to-many relationships between concrete and abstract syntax, though there are opportunities to do so. For example, BPMN concrete syntax could be used with BPMN abstract syntax or with UML abstract syntax extended by the UML Profile for BPMN Processes [38]. BPMN graphics combined with an extended UML abstract syntax would enable system operational requirements and other behaviors external to a system to be specified by operators in a more accessible notation (BPMN), but still be available to design engineers in a more technical language (UML) [7].

This paper gives definers of graphical languages a way to address the challenges above, via many-to-many relation-

[1] Precursors to metamodels in academic work used links between abstract syntax tree elements that are not hierarchically related [23,24].

[2] An earlier modeling language in the product modeling community specified graphical and textual concrete syntax for the same language concepts [9], with metamodels added later at OMG [26].

[3] It also did not separate user-controllable graphical information, such as position and size, from standards-defined information, such as shapes, resulting in redundant inclusion of standards information in interchange files; see Sect. 3.

[4] Also see Footnote 2.

ships between concrete and abstract syntax, in Sects. 4 and 5.

Other work related to separately modeling concrete and abstract syntax is not concerned with the challenges above. Early work on model–view–controllers enabled multiple visualizations of the same underlying model, but programmatically related them, rather than modeling concrete syntax explicitly and giving rules for its relation to abstract syntax, as needed for specifying standard graphical languages [39–41]. Another effort enables a single element of graphical concrete syntax for a group of related abstract syntax elements, rather than for separate abstract syntaxes as in this paper, and addresses dynamic, bidirectional synchronization of concrete and abstract syntax [11]. DD also supports single elements of graphical concrete syntax for groups of related abstract syntax elements and can potentially be used in synchronization architectures developed in that work. Other efforts propose unified models of concrete and abstract syntax [12,13]. This reduces redundancy and inconsistency between the two kinds of syntax, but eliminates the benefits of separating them (see discussion about syntactic redundancy in DD at the end of Sect. 3.1). Another approach to reducing redundancy between concrete and abstract syntax is to automatically produce abstract syntax from concrete [14]. This technique would have wider application using a standard model of concrete syntax, such as DD. Other work upgrades textual parser generators to use abstract syntax [15–17]. The efforts cited here are helpful in various ways, but do not address the two gaps described above and in Sect. 1.

3 Concrete graphical syntax using Diagram Definition

Cases examined in this paper use the MOF subset of UML for specifying abstract syntax (see Sect. 2) and DD for specifying concrete graphical ("node and arc") syntax. DD has two parts:

- Diagram Interchange (DI) is for syntax that modelers control, such as position of nodes and routing of lines, and notational options. This is captured for interchange between graphical tools.
- Diagram Graphics (DG) is for syntax that modelers do not control, such as the kinds of shapes and line styles defined by language specifications. This is not interchanged because it is the same across all tools conforming to a language.

DI and DG are shown by bold-outlined rectangles in Fig. 1 (adapted from a figure developed by the authors in [3]). They both provide a metamodel defined in MOF, which is instantiated when graphical languages are used, as described in Sects. 3.1 and 3.2. Mappings between DI and DG, shown as the bold arrow on the lower middle of Fig. 1, are covered in Section 3.2.

3.1 Diagram Interchange

Language specifications specialize elements of the DI metamodel for their particular kinds of nodes and arcs (Shapes

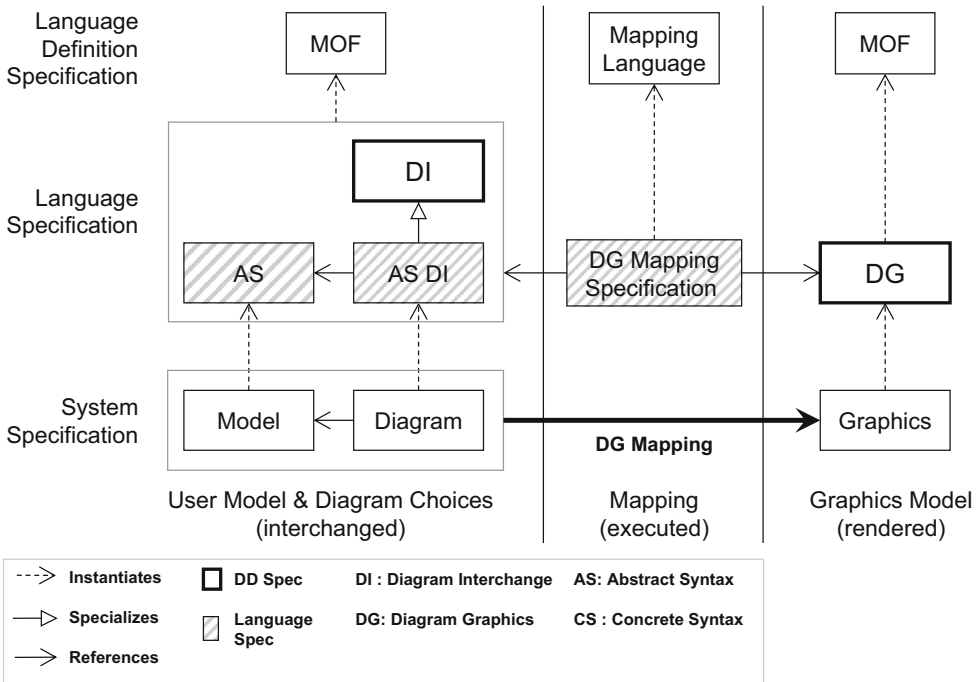


Fig. 1 Diagram definition architecture, adapted from [3]

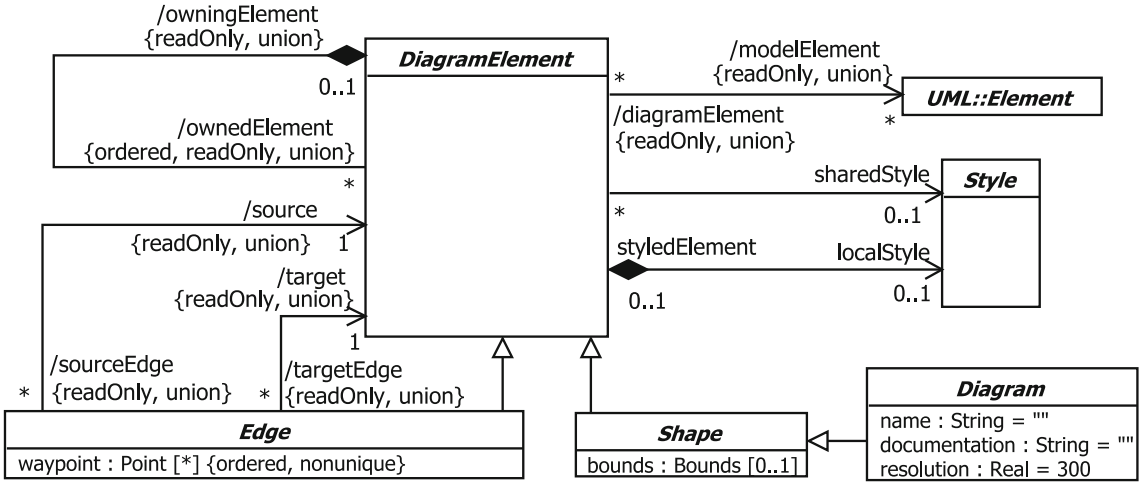


Fig. 2 Diagram Interchange metamodel

and Edges in DI), as shown by the shaded rectangle labeled "AS DI" on the left in Fig. 1. These specializations identify elements of the language's abstract syntax they notate, as shown by the arrow from AS DI to the shaded rectangle labeled "AS". The specializations also introduce additional modeler-controlled options for the concrete syntax, such as alternative icons or presentation styles. Examples of these are given for UML and BPMN in Sects. 4.1 and 5.1, respectively.

Constructing models using graphical languages causes instances to be created from their abstract syntaxes and DI specializations, linked together, as shown by the rectangles labeled "Model" and "Diagram" in Fig. 1," respectively, and the arrow between them. The resulting instances of DI specializations can be sent to other graphical tools, along with the corresponding instances of abstract syntax, to tell other tools where nodes and arcs are to be placed and routed, how their appearance should be adjusted for any modeler-controlled options the concrete syntax has, and which instances of abstract syntax they notate.

Figure 2 shows the DI metamodel. Its most abstract element is DiagramElement, which generalizes all the others. Diagram elements can identify the elements of abstract syntax they notate (via modelElement), which can be any element defined in UML, including its MOF subset. Some diagram elements might not refer to abstract syntax, while others might refer to more than one element of abstract syntax. Examples of these cases are given in Sects. 4.2 and 5.2, respectively.

Diagram elements are laid out visually based on whether they are shapes or edges. Shapes are positioned within bounding rectangles, while edges are displayed as a series of line segments specified by a list of waypoints going from sourceElements to targetElements. Diagram elements can be nested recursively (via ownedElement), most commonly within Diagrams, which are shapes that establish new coordinate systems for their nested elements. The top-left corner of a diagram is the origin. All locations and sizes are given in display device units, typically pixels. Overlapping diagram elements are displayed on top of elements in which they are nested. Diagram elements can specify Styles (for modeler-defined visual properties such as colors and fonts), either inherited from an element they are nested in, shared with other elements (via sharedStyle), or owned directly (via localStyle). Other aspects of Fig. 2 are described in the DD specification [3].

The DI specification does not restrict how languages specialize DI elements. They might minimize the number of specializations, to reduce redundancy with abstract syntax, or they might include specializations closely mirroring the abstract syntax, to clarify how the language DI is to be used with abstract syntax, or somewhere in between these two approaches. For example, language specifications might have only one shape specialization that has all modeler-controlled options, even though only some of the options apply in particular uses, depending on which abstract syntax element is being notated. At the other extreme, language specifications might have as many shape and edge specializations as there are in its abstract syntax, with modeler-controlled options introduced only in specializations that support them. In between these approaches, language specifications might have a moderate number of shape and edge specializations, with some partitioning of modeler-controlled options over them, but not a separate specialization for each abstract syntax element or each modeler-controlled option (see UML's DI specializations in Sect. 4.1).

3.2 Diagram Graphics

Language specifications use DG to describe how instances of their DI specializations are displayed. This has two parts:

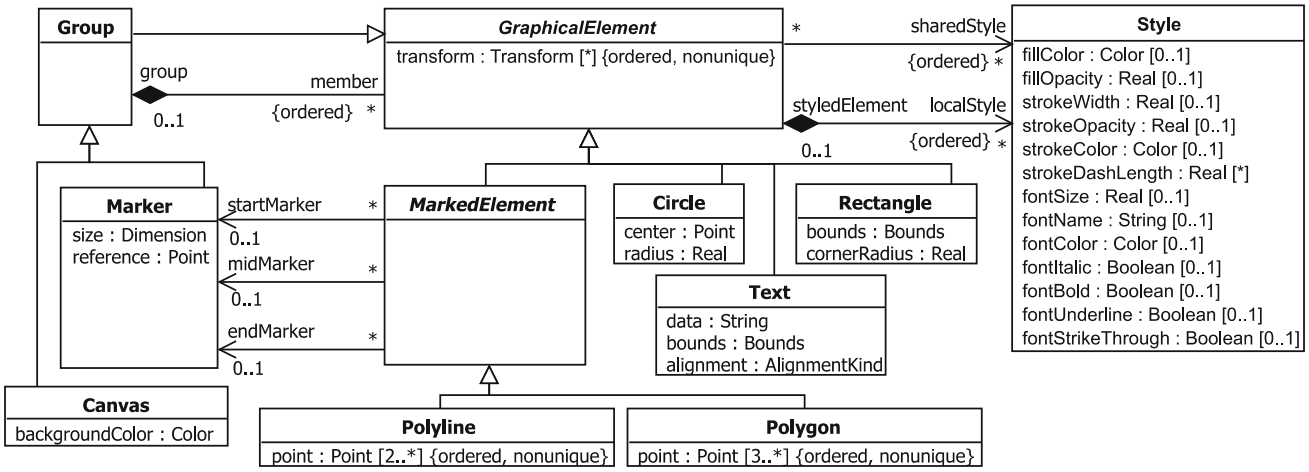


Fig. 3 Diagram Graphics metamodel (excerpt)

- The DG metamodel for graphics, such as rectangles and lines, similar to Scalable Vector Graphics (SVG) [42], as shown by the bold-outlined rectangle labeled "DG" on the right in Fig. 1.
- Rules for instantiating DG elements based on instances of DI specializations, as shown by the shaded rectangle labeled "DG Mapping Specification" in the middle of Fig. 1 (for the rules), and the arrow labeled "DG Mapping" to the rectangle "Graphics" on the right in Fig. 1 (for instantiation of DG).

Figure 3 shows a portion of the DG metamodel. The most abstract element is GraphicalElement, which covers elements that can group other graphical elements recursively (via member) and "primitives" that cannot, such as Rectangles, Circles, and Text. At the top of recursive groupings are Canvases, the graphical analogs of diagrams. Some primitive elements are defined as sets of points connected by lines, such as Polygons and Polylines. Primitives may be decorated with groups of elements at their start, middle and end points (Markers). Graphical elements can specify Styles (for predefined visual properties such as fillColor and fontName), either inherited from groups they belong to, shared with other elements (via sharedStyle), or owned directly (via localStyle).

Language specifications give rules for instantiating DG elements, taking instances of their DI specializations as input. For example, a language might have an abstract element notated as a rectangle. Rules find the intended bounds of these rectangles from instances of DI specializations referring to instances of that abstract syntax element, then instantiate Rectangle in the DG metamodel, and set its position and size according to intended bounds. Languages specify mappings from their DI specializations to DG to give a more precise specification of their notation, as compared to only describing notation informally.

Rules for instantiating DG elements in this paper are defined using the Query View Transform Operational language (QVTo) [43], though any transformation language operating on instances of MOF-based metamodels or their interchange formats could be used (explanation of QVTo is limited to salient aspects as needed, for brevity). Listing 1 gives a general DI to DG transformation filled out in the rest of the paper for the cases examined. First it declares the kind of models input to and output from the transformation, specifically models containing instances of DI or DG elements or their specializations, including those given in the rest of the paper. The body of the transformation selects out instances of Diagram in the input model with the QVTo objectsOfType operation and then uses the QVTo map operation to apply toGraphics mappings to them, defined just below the transformation. Mappings operate on any instance in the input model of the transformation that are of the type given before the double colon in the mapping definition. In Listing 1, the mapping operates on any kind of diagram defined with DI. Mappings create a new instance of the type appearing after the single colon, a canvas in this case. The body of the mapping in Listing 1 adds graphical elements to the canvas created by toGraphicsTop mappings applied to each element in the diagram. These mappings are defined by cases examined in Sects. 4.2 and 5.2.

```
modeltype DI uses " http://www.omg.org/spec/DD/20131001/DI.xmi";
modeltype DG uses " http://www.omg.org/spec/DD/20131001/DG.xmi";

transformation DIToDG(in adi : DI, out DG);
main() { adi.objectsOfType->selectByKind(Diagram)->
           map toGraphics(); }

mapping Diagram::toGraphics() : Canvas {
member += self.ownedElement->map toGraphicsTop(); }
```

Listing 1 Instantiating DG from instances of DI

4 Multiple notations on a single metamodel

This section examines cases of multiple graphical notations for the same metamodel (see first numbered bullet in Sect. 1), drawn from UML. UML includes DI specializations, developed by the authors and standardized by OMG [4,5], but not mappings to DG. Section 4.1 outlines UML's DI specializations, including their design principles. Section 4.2 shows how they support cases of alternative graphical notations for the same metamodel elements in UML, and how these can be mapped to DG.

4.1 UML Diagram Interchange

In defining UML's DI specializations (UML DI), the authors chose an approach producing a moderate number of specialized classes, described at the end of Sect. 3.1. In particular, UML DI specializes DI to define:

- Instantiable elements corresponding to DI's non-instantiable elements, such as UML DI's UMLDiagram specializing DI's Diagram, as shown in Fig. 4. This includes specializing associations, such as redefinition of properties for nesting (owned/owningElement).
- UML's diagram taxonomy, which specializes UMLDiagram into various diagram types, including abstractions such as UMLBehaviorDiagram generalizing concrete diagram types, such as UMLActivityDiagram, UMLStateMachineDiagram, and UMLInteractionDiagram, as shown in Fig. 5. Diagrams introduce properties at the appropriate level, for example, to specify whether diagrams have frames on UMLDiagram, and the kind of interaction diagram on UMLInteractionDiagram. Some diagrams refer to specific elements of abstract syntax they notate, such as state machine diagrams referring to state machines (omitted from Fig. 5 for brevity).
- Abstractions for characteristics used in multiple parts of the model, such as UMLDiagramWithAssociation to specify navigation arrow styles in Fig. 5 and UMLCompartmentableShape generalizing elements that can have compartments on the right in Fig. 6 (properties omitted for brevity). These abstractions are also used in structural diagrams and elements (omitted for brevity).
- Labels for displaying text, as shown on the left in Fig. 6. Specialized labels indicate the particular aspects of abstract syntax the text is about when this is ambiguous or cumbersome to determine otherwise. They also enable receivers of UML DI files to update displayed text when portions of the underlying abstract syntax instances change. For example, UMLNameLabel is for labels showing names of abstract syntax elements, enabling tools that receive UML DI files to update displayed text rendered when names of elements change. UML DI does not model substring structure of text, leaving this to other parts of UML, which typically specify textual notation using BNF (see Sect. 2).

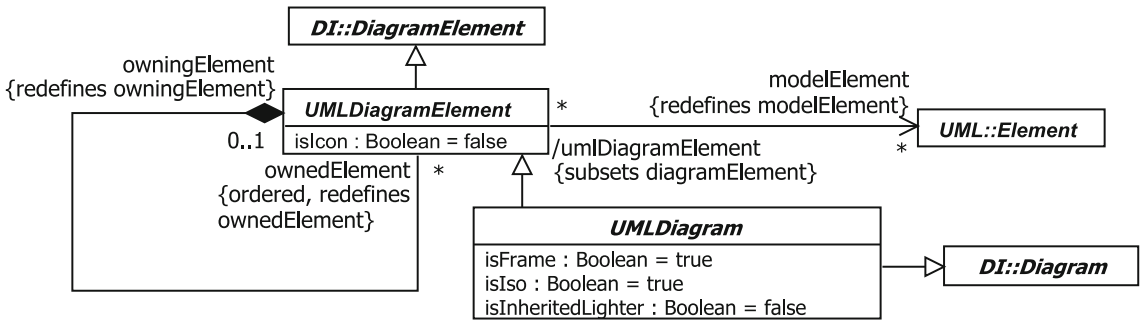


Fig. 4 UML diagrams and elements (excerpt)

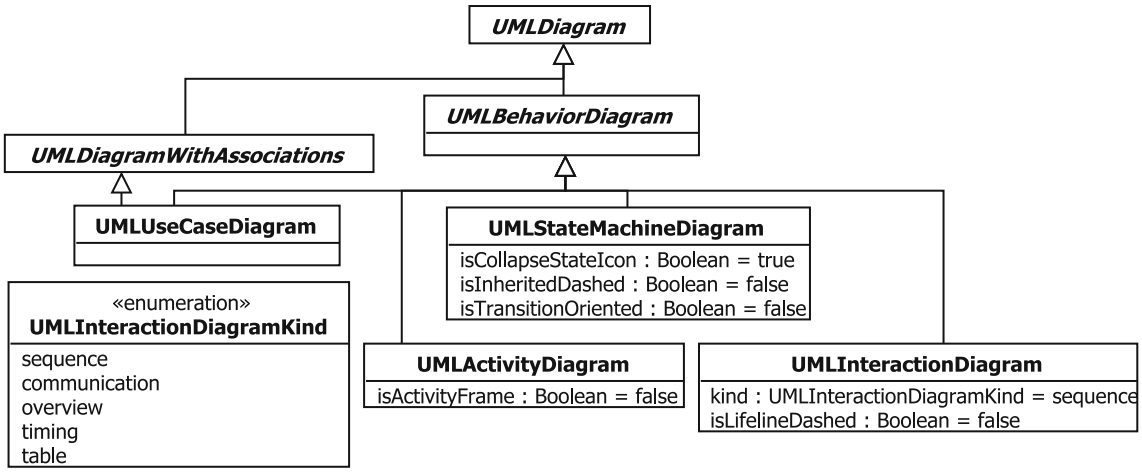


Fig. 5 UML diagram taxonomy (excerpt)

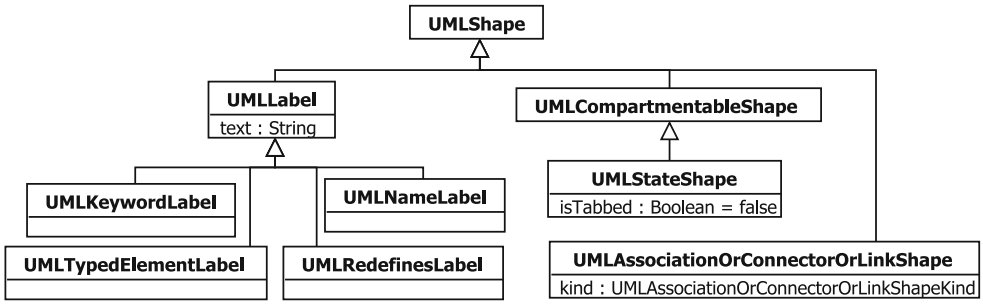


Fig. 6 UML shapes (excerpt)

- Elements corresponding to specific abstract syntax in limited cases, such as UMLStateShape in Fig. 6, to specify whether UML's state notation uses its tabbing option and AssociationOrConnectorOrLinkShape for UML's triangle and diagram notation options on associations, connectors, and links.
- Abstract properties that apply only in some specializations, in limited cases, such as the isIcon property of UMLDiagramElement, indicating whether to display graphical modifications or adornments for some notations. For example, UML's realization dependency arrow can be shown headed with a hollow triangle or with a regular arrowhead and a label, depending on whether the value of isIcon is true or false, respectively.

UML DI avoids specializations other than the above. In particular, it does not model display options typically handled by tools, such as which features should be shown on which elements, or colors and shading. For example, tools might enable modelers to indicate that only properties are shown on class rectangles, not behavioral features. As features are added to classes, the tool will display properties, but not operations. UML DI interchange files will specify the properties displayed, but not the display constraint that only properties should be shown. This is because the receiver might not want to be constrained in this way and because there are so many of these options supported in tools that it would overwhelm the standard to cover them all.[5] For the same reasons, UML DI also avoids style modeling, such as colors and shading, but does include a style for font names and sizes, because these affect layout.

4.2 Multiple UML notations on a single metamodel

UML defines multiple graphical notations for portions of its metamodel, enabling automated tools to operate on those portions, even when they are presented with different visual emphasis and organization. For example, properties can be notated as text inside class rectangles, as labeled lines between class rectangles, or as rectangles themselves within an entire diagram for the class. Text within rectangles is easier to scan, but makes it more difficult to see interconnections between classes, whereas lines between rectangles highlight their interconnections, but spread properties over a wider visual space. Notating properties as rectangles enable connections between properties to be shown, but are easily confused with associations between classes. Modelers can

[5] Tools might infer display constraints from UML DI files, for example, when only properties are shown on classes.

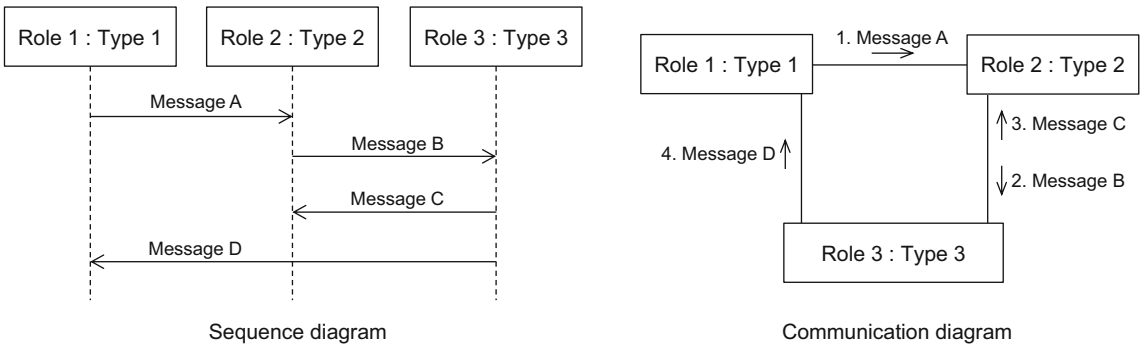


Fig. 7 UML sequence and communication diagrams

choose among these property notations for their intended audience and use the same automated tools on the shared metamodel, such as model checkers and code generators.

In particular, UML has several notational variations for specifying messages sent between objects, collectively called *interaction diagrams*:

- *Sequence diagrams* highlight message order more compactly than other interaction diagrams. They show messages as arrows laid out in the order in which messages are to be sent, between dashed lines notating objects.
- *Communication diagrams* are the only kind of interaction diagrams that show connections between objects that messages flow along. They notate objects as rectangles linked by lines for structural connections between them. Messages are shown as small unattached arrows next to connecting lines, labeled according to the order in which they are sent.
- *Interaction overview diagrams* highlight message order the most, but are the least compact kind of interaction diagram. They have a flowchart style, with nodes containing interaction diagram fragments showing messages.
- *Timing diagrams* highlight state changes over time more than other interaction diagrams. One kind uses graphs with object states on one axis and time and messages sent on the other, while the other just lays out states on a time axis.

For example, Fig. 7 shows an interaction with three *lifelines*, which are participants or roles in the interaction. In sequence diagrams, these are notated as lines with rectangular headers, as on the left, and in communication diagrams as just rectangles, on the right. Lifelines are labeled with their role name to the left of a colon and the kind of things playing the role to the right. Sequence diagrams specify the order in which messages occur by the order that message arrows are laid out, top to bottom. Figure 7 shows a message from the leftmost lifeline sent before the exchange between the right two lifelines and a message from the rightmost lifeline to the leftmost sent last. Modelers mainly interested in message order probably prefer sequence diagrams for their clarity and compactness. Communication diagrams specify message order by numbered labels on unattached message arrows. They are the only interaction diagrams that notate structural connections between lifelines. Message arrows near connecting lines in communication diagrams show messages that flow over those structural connections. Modelers concerned with the relationship between structural connections and messages use communication diagrams to see this information graphically.

All UML interaction notations show the same portion of the UML metamodel for the most part, enabling the same interaction to be notated in ways most useful for modelers working with it. Figure 8 shows instances of UML DI for the first two messages in Fig. 7 on the left and right, linked to the same instances of UML metamodel elements in the middle. The figure uses UML's notation for *instance specifications*, which are rectangles with underlined labels, where labels give the name of the UML DI or UML metamodel element instantiated to the right of a colon and the instance name to the left (properties with no values and some instance names are omitted for brevity). Lines between the rectangles correspond to relationships between UML DI instances (see Sect. 4.1) and between those and UML metamodel instances (via modelElement). Elements for text in Fig. 7 are omitted from Fig. 8 for brevity.

The UML metamodel instances in Fig. 8 are mostly referenced by two UML DI instances, one each from the sequence and communications diagrams, reflecting the many-to-one relationship between concrete and abstract syntax in this case. Each metamodel instance for a lifeline is referenced by three UML DI elements, one from the communication diagram and two from the sequence diagram, a shape for the header rectangle and an edge for the vertical line (these only have a source, no target). Metamodel instances for ends of messages (*message occurrence specifications*) are not notated and have no corresponding UML DI instances in either diagram. The vertical order of message arrows in sequence diagrams is reflected by *general orderings* from earlier message occurrences to later (not shown for brevity). In Fig. 8, a general ordering would be shown between the

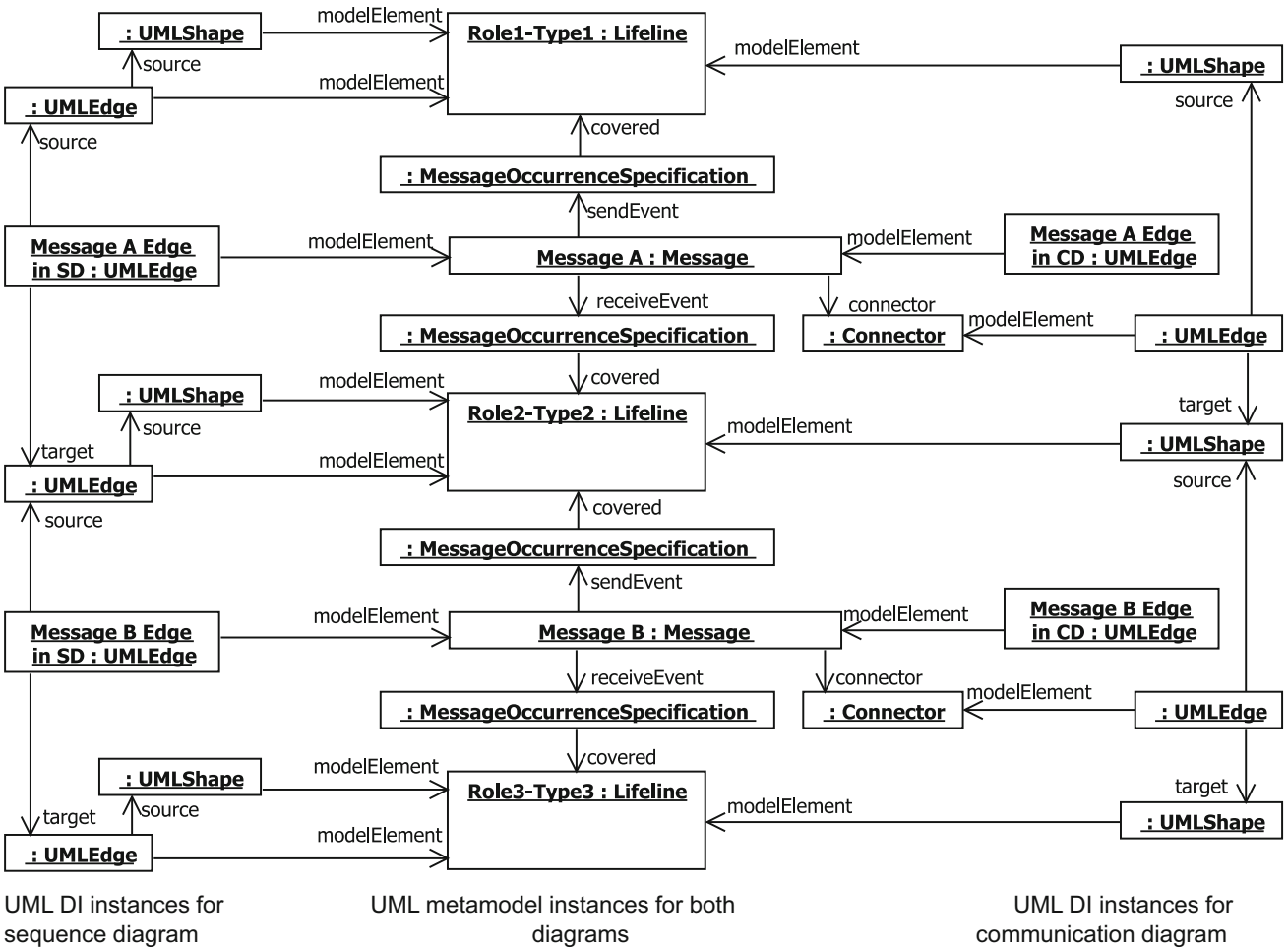


Fig. 8 Instances of UML DI and UML metamodel for portion of Fig. 7

receiveEvent of Message A and the sendEvent of Message B, indicating that Message B that is sent after Message A is received. In communication diagrams, edges between lifeline shapes are linked to metamodel instances for connectors (the only diagram that uses them, see bullets at the beginning of this section). Messages link to connectors they flow across (links between metamodel instances for connectors and lifelines are omitted for brevity). Metamodel instances for messages are notated as unattached message arrow edges (these have no source or target).

dividing them into two kinds, those that identify instances of the UML metamodel (via modelElement, see Sect. 4.1) and those that do not. The if statement applies toGraphics mappings to UML metamodel instances when they are present, otherwise to diagram UML DI instances themselves. When toGraphics is applied to metamodel instances, the UML DI instance is given as an argument; otherwise, it is applied directly to the UML DI instance without arguments. In both cases, the result of the mapping is a new DG graphical element that is added to the canvas created in Listing 1.

```
mapping UMLDiagramElement::toGraphicsTop() : GraphicalElement {
  var me := self.modelElement;
  if (me->isEmpty()) result := self.map toGraphics();
  else result := me.map toGraphics(self); }
```

Listing 2 Instantiating DG from instances of UML DI

Listing 2 defines a mapping used by the general DI to DG transformation in Listing 1 in Sect. 3.2 (toGraphicsTop). It operates on elements of UML DI (UMLDiagramElements),

Listing 3 defines toGraphics mappings for use by Listing 2, producing instances of DG for rendering sequence and communication diagrams like those in Fig. 7 from

```
mapping Lifeline::toGraphics (de:UMLDiagramElement) : Rectangle
  when {de.oclIsKindOf(UMLShape)and
        not de.interactionDiagramKind = timing)} {
de.oclAsTypeOf(UMLShape).setRectangle (result); }

mapping Lifeline::toGraphics (de:UMLDiagramElement) : Polyline
  when {de.oclIsKindOf(UMLEdge)} {
de.oclAsTypeOf(UMLEdge).setPolyline (result) ;
sharedStyle := umlDashedLine; }

mapping Message::ToGraphics (de:UMLDiagramElement) : Polyline {
de.oclAsTypeOf(UMLEdge).setPolyline (result) ;
endMarker := umlArrowHead; }

mapping Connector::ToGraphics (de:UMLDiagramElement) : Polyline {
de.oclAsTypeOf(UMLEdge).setPolyline (result) ; }
```

Listing 3 Instantiating DG for UML sequence and communication diagrams

instances of UML DI like those in Fig. 8, as well as portions of overview diagrams. The mappings operate on UML metamodel instances such as lifelines, with UML DI instances as arguments, because all the diagram elements in sequence and communication diagrams identify UML metamodel instances (via modelElement). QVTo when clauses specify conditions to meet for a mapping to apply. For example, the first mapping in Listing 3 creates rectangles notating lifelines, as in Fig. 7. It only applies for diagram elements that are UMLShapes in diagrams other than timing. When these conditions are met, the mapping creates a DG rectangle and uses the setRectangle query to position it on the canvas in the location given by the diagram element. The definition of setRectangle is given in Listing 4. The second mapping in Listing 3 has a similar effect for the line portions of lifeline notation in sequence diagrams, using the setPolyline query and umlDashedLine style defined in Listing 4. The rest of the mappings apply to other kinds of UML metamodel instances and work in a similar way. Mappings for text are omitted for brevity.

Figure 9 shows the first two messages in Fig. 7 in an overview diagram on the left and all four messages in a timing diagram on the right. Interaction overview diagrams show each message in its own embedded diagram, with arrows between the diagrams for the order in which they occur, as in flowcharts. Modelers interested in decision logic or parallelism between messages might prefer overview diagrams even though they are less compact than other interaction diagrams. Timing diagrams graph states of objects over time. Time appears on a horizontal axis and states appear either vertically, as in the top and middle right of Fig. 9, or laid out horizontally on the time axis, as in the bottom right. Messages causing state changes are notated by labeling the points at which they occur. Other interaction diagrams support state symbols, but modelers concerned mainly with state changes would prefer timing diagrams over the others. They can show more information than the other interaction diagrams, but the relationships between the elements are more difficult to see. The sequence diagram in Fig. 7 and the timing diagram in Fig. 9 assume messages are transferred

```
query UMLDiagramElement::interactionDiagramKind () :
  UMLInteractionDiagramKind =
  self.owningElement->selectByKind(UMLInteractionDiagram).at(1).kind;
query Shape::setRectangle (r : Rectangle) = r.bounds := self.bounds;

query Edge::setPolyline (pl : Polyline) = pl.point := self.waypoint;

property umlDashedLine = object DG::Style {
strokeDashLength := Sequence {2, 2}; };

property umlArrowHead = object Marker {
size := object Dimension {width:=10; height:=10};
reference := object Point {width:=10; height:=5};
member += object PolyLine { point += object Point{x:=0; y:=0};
                            point += reference;
                            point += object Point{x:=0; y} }; };
```

Listing 4 Support for mappings in Listing 3

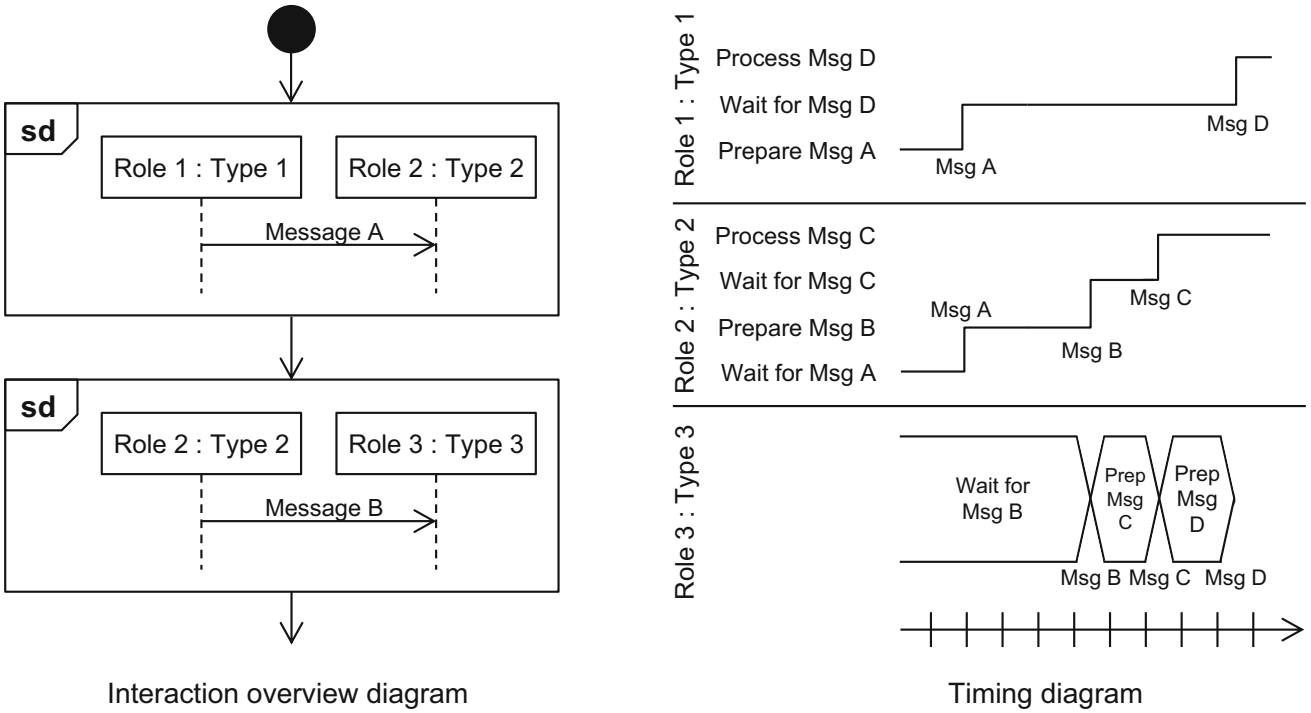


Fig. 9 UML interaction overview and timing diagrams

instantly, as indicated by the horizontal message arrows in Fig. 7 and the synchronization of sending and receiving in the timing diagram in Fig. 9. Time taken for message transfer can be shown with angled message arrows in sequence diagrams and unsynchronized sends and receives in timing diagrams.

Figure 10 shows instances of UML DI on the left and right for overview and timing diagrams in Fig. 9, respectively, linked to the same UML metamodel instances as the middle as Fig. 8 (except connectors are removed, because they are not in Fig. 9, and states are added for the timing diagram). Overview diagram instances on the left are in two columns, the leftmost being the embedded diagram shapes and the edge between them, and just to the right other shapes and edges contained by the embedded diagram shapes (the edge and shape on the top left are the initial node and edge leading to the top embedded diagram shape). Embedded diagram shapes do not refer to metamodel instances because the shapes do not represent reusable interactions and they are purely graphical elements. The same is true of the shape and edge at the top left, which indicate the beginning of the interaction. The edge between embedded diagram shapes refers to a general ordering (not shown for brevity) between the top two message occurrence specifications in the middle, which are the tail and head ends of Message A and Message B, respectively. Shapes and edges owned by the embedded diagram shapes link to metamodel instances in the same way as those in sequence diagrams, see the left of Fig. 8. Elements for text in the overview diagram are omitted from Fig. 10 for brevity.

Most of the timing diagram instances on the right of Fig. 10 are linked to the same metamodel instances as the overview diagram instances on the left. A special kind of label is used for lifelines (UMLTypeElementLabels), which has a colon separating the name or role of an element from its type (these are omitted on the left for brevity). Most of the other labels are paired with an edge (line segment with no arrowheads) referring to the same metamodel instance as the label. For example, the Prepare Msg A state instance has a label and edge referring to it. The label shows the name of the state (because it is a UMLNamedElementLabel), and the edge shows the time interval during which the object is in that state (the horizontal line at the same level as the label). Links between metamodel instances for states and lifelines are omitted for brevity. Non-name labels refer to message occurrences that also have an edge referring to them. For example, Role 1 Msg A Lbl is for the "Msg A" string in the upper section of the timing diagram in Fig. 9 that refers to a message occurrence that also has an edge referring to it (the string is recorded in the label, but omitted from Fig. 10 for brevity). This edge is for the vertical line segment above the "Msg A" string, notating the receipt of Message A. The edge is the target of the horizontal edge leading to it (for the Prepare Msg A state), because the line segments

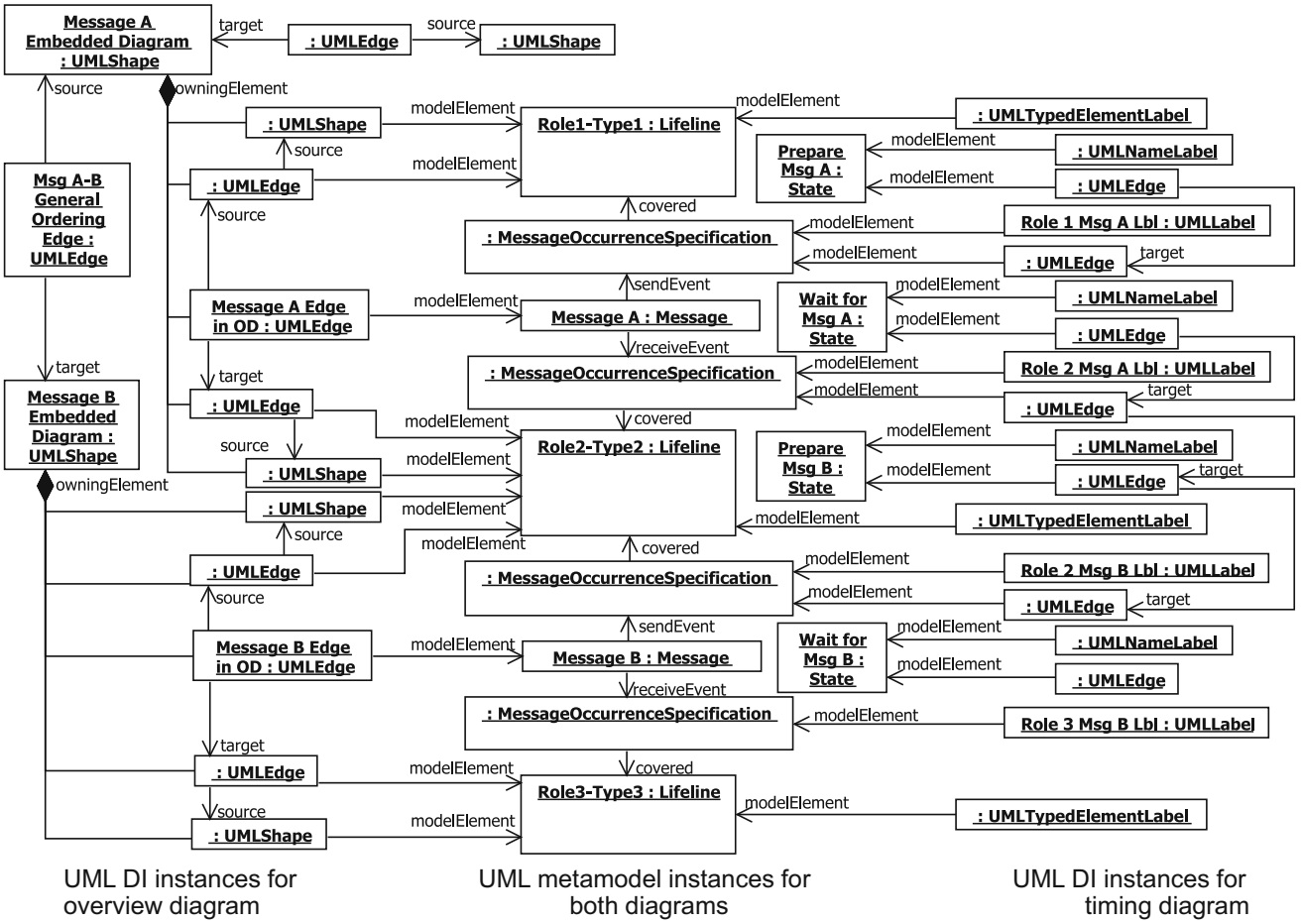


Fig. 10 Instances of UML DI and UML metamodel for Fig. 9

are joined. The same patterns of UML DI and metamodel instance are used for the remaining two timing roles in the middle and bottom right of Fig. 9, except that the third role on the bottom does not notate message occurrence specifications (the message occurrence at the bottom right is referred to only by a label, no edge). The notation of the third role shows states laid out on the timeline, with their intervals indicated by hexagons. The horizontal order of state and message occurrence elements in timing diagrams is reflected by general orderings from earlier message occurrences to later (not shown in Fig. 10 for brevity). Edges for lines between roles and the time axis in Fig. 9 are also omitted from Fig. 10 for brevity, as is the edge for the arrow on the lower right of Fig. 9.

Listing 5 defines toGraphics mappings for use by Listing 2, producing instances of DG for rendering overview diagrams, like the one on the left in Fig. 9, from instances of UML DI like those on the left in Fig. 10. The mappings cover the portions of overview diagrams not addressed by Listing 3, primarily diagram elements that do not identify UML metamodel instances. For example, the first two mappings in Listing 5 are for the filled circle and arrow at the top left of Fig. 9, respectively, which indicate the start of the interaction. The first uses setCircle to position the circle on the canvas and blackFillStyle to fill it, see Listing 6. The third and fourth mappings are for the embedded diagram frames and their heading in the middle left of Fig. 9, which in this case are purely visual, without an interaction defined in metamodel instances. The fourth mapping creates a group for the heading, containing a label and the line around it. It uses setText to assign and position of the string displayed, see Listing 6. The definition of setHeadingPolyLine is omitted for brevity. The last mapping, the only one based on metamodel instances, is for the arrow between the two embedded diagrams, which notates the temporal ordering between the messages in those diagrams (this is the general ordering omitted from Fig. 10).

```
mapping UMLShape::toGraphics () : Circle
  when {self.interactionDiagramKind = overview and
        self.targetEdge->isEmpty()} {
self.setCircle(result);
sharedStyle := blackFillStyle; }

mapping UMLEdge::toGraphics () : Polyline
  when {self.interactionDiagramKind = overview} {
self.setPolyline(result); }; }

mapping UMLShape::toGraphics () : Rectangle
  when {self.interactionDiagramKind = overview and
        self.targetEdge->notEmpty()} {
s.setRectangle(result); }

mapping UMLLabel::toGraphics () : Group
  when {self.interactionDiagramKind = overview} {
var t := object Text;
self.setText(t);
member += t;
var pl := object Polyline;
self.setHeadingPolyLine(pl);
member += pl; }

mapping GeneralOrdering::toGraphics (de : BPMNDiagramElement) :
  Polyline when {de.interactionDiagramKind = overview} {
de.setPolyline(result);
result.endMarker := umlArrowHead; }
```

Listing 5 Instantiating DG for UML interaction overview diagrams

```
query Bounds::center () : Point =
object Point{self.x + self.width/2; self.y + self.height/2};

query Shape::setCircle (c : Circle) {
  var b := self.bounds;
  c.center := b.center();
  c.radius := if b.width<b.height b.width/2
              else b.height/2 ; }

query UMLLabel::setText (t : Text) {
  t.bounds := self.bounds;
  t.data := self.text;
  t.alignment := center;}

property blackFillStyle = object DG::Style {
  fillColor := object Color {red:=0;green:=0;blue:=0}; }
```

Listing 6 Support for mappings in Listing 5

Listing 7 defines toGraphics mappings producing instances of DG for rendering timing diagrams, like the one on the right in Fig. 9, from instances of UML DI like those on the right in Fig. 9. Timing diagrams are very different from other interaction diagrams and do not share DG mappings with them. For example, timing lifelines are just text, ori-

ented vertically on the left side of the diagram, with no shapes or edges as in the other interaction diagrams. This aspect is addressed by the first mapping. It uses rotateVertical to display vertical text, which adds a transform to the graphic, see Listing 8. Regions of the diagram labeled by vertical lifeline text are separated by line segments, with a timing axis below the bottom lifeline, none of which refer to metamodel instances. These are covered by the second mapping, which creates lines for these edges, adding an arrowhead for the timing axis when detected by isTimingAxis, see Listing 8. States appear in two forms, first as horizontal text on the left side next to lifeline text. This form is at the same height as its second form, horizontal lines indicating the time during which a lifeline's object is in that state (see the third and fourth mappings). Message occurrence specifications also appear in two forms: first as the name of a message at the point it is sent or received and second as a vertical line at the same point. These are handled by the last two mappings.

```
mapping Lifeline::toGraphics (de:UMLDiagramElement) : Text
  when {de.interactionDiagramKind = timing
        and de.oclIsKindOf (UMLLabel)} {
de.oclAsTypeOf(UMLLabel).setText(result);
result.rotateVertical(); }

mapping UMLEdge::toGraphics () : Polyline
  when {self.interactionDiagramKind = timing} {
self.setPolyline(result);
if (isTimingAxis(self)) {e.endMarker := umlArrowHead; }; }

mapping State::toGraphics (de:UMLDiagramElement) : Text
  when {de.interactionDiagramKind = timing
        and de.oclIsKindOf (UMLLabel)} {
de.oclAsTypeOf(UMLLabel).setText(result); }

mapping State::toGraphics (de:UMLDiagramElement) : Polyline
  when {de.interactionDiagramKind = timing
        and de.oclIsKindOf (UMLEdge)} {
de.oclAsTypeOf(UMLEdge).setPolyline (result) ; }

mapping MessageOccurrenceSpecification::toGraphics
  (de:UMLDiagramElement) : Text {
  when {de.oclIsKindOf (UMLLabel)} {
de.oclAsTypeOf(UMLLabel).setText (result); }
mapping MessageOccurrenceSpecification::toGraphics
  (de:UMLDiagramElement) : Line {
  when {de.oclIsKindOf (UMLEdge)} {
de.oclAsTypeOf(UMLEdge).setPolyline (result); }
```

Listing 7 Instantiating DG for UML timing diagrams

```
query GraphicalElement::rotateVertical () {
  var h := self.bounds.height;
  self.bounds.height: = self.bounds.width;
  self.bounds.width: = h;
  self.bounds.y := self.bounds.y + h;
  self.transform += object Rotate{
     angle := -90; center := object Point{x:=self.bounds.x;
                                          y:=self.bounds.y}};}

query UMLEdge::isTimingAxis () : Boolean {
  var selfy := self.waypoint.at(1).y;
  return self.isHorizontal and
         not self.owningElement.ownedElement->
               selectbyKind(UMLEdge)->
                 exists ( e | isHorizontal (e) and
                              e.waypoint.at(1).y >= selfy ); }

query UMLEdge::isHorizontal() : Boolean =
  self.waypoint->first().y = self.waypoint->last().y;
```

Listing 8 Support for mappings in Listing 7

5 Single notation on multiple metamodels

This section examines a case of the same graphical notation for multiple metamodels (see second numbered bullet in Sect. 1), drawn from BPMN and UML. BPMN includes DI specializations, developed by a team including one of the authors (Elaasar), but not mappings to DG. Section 5.1 outlines BPMN's DI specializations, including their design principles. Section 5.2 shows how the specializations support BPMN notation for the BPMN and UML metamodels, and how they can be mapped to DG.

5.1 BPMN Diagram Interchange

BPMN includes an early version of DI, slightly different from the one OMG standardized. This section presents a BPMN DI specialized from the standard DI, with some restrictions loosened to support the case examined in Sect. 5.2.

BPMN has a small number of modeler-controlled notation options, especially compared to UML. In defining BPMN's DI specializations, the author and his team chose to have all modeler-controlled options defined on very few DI specializations, an approach described at the end of Sect. 3.1, and in particular not to include a diagram taxonomy specializing DI's Diagram for the various types of BPMN diagram. BPMN DI specializes DI to define:

- Instantiable elements corresponding to DI's non-instantiable elements, such as BPMN DI's BPMNDiagram specializing DI's Diagram, as shown in Fig. 11, and BPMNShape and BPMNEdge, as shown in Fig. 12.
- Labels and label styles for text, one specialization for each, BPMNLabel and BPMNLabelStyle, respectively, as shown in Fig. 13.

All modeler-controlled notation options for shapes and edges are defined on BPMNShape and BPMNEdge, with no further specializations, applying only when particular abstract syntax elements are being notated. For example, the property isHorizontal on BPMNShape only applies when

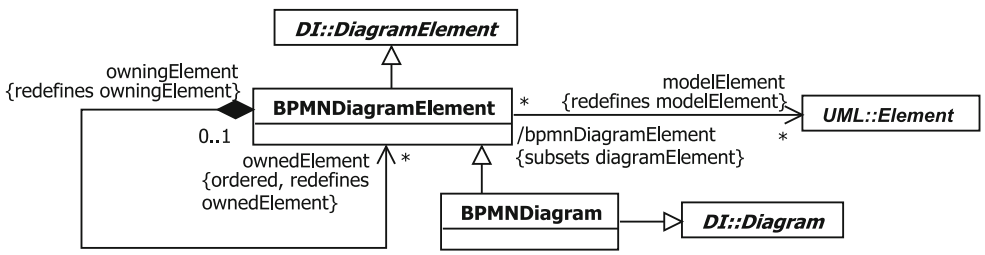


Fig. 11 BPMN diagrams and elements

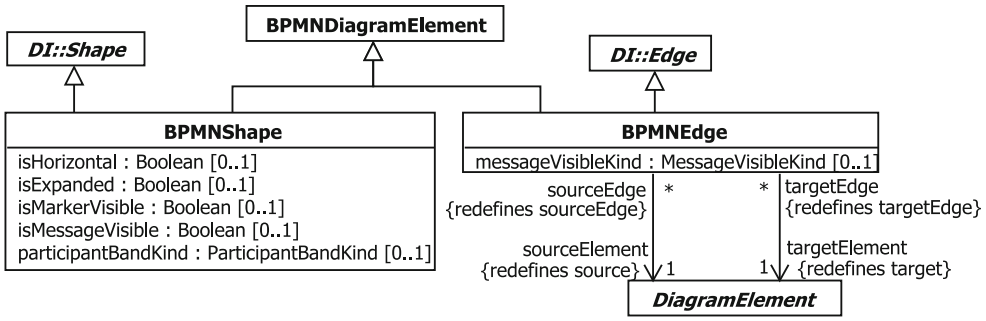


Fig. 12 BPMN shapes and edges

Fig. 13 BPMN labels and styles

notating process participants as pool or lane rectangles, to indicate whether their long axis should be horizontal or vertical. The messageVisibleKind property on BPMNEdge only applies when notating messages, to indicate whether a message icon appears overlaid on arrows. The others have similarly restricted application, as described in the BPMN specification.

The BPMN DI presented here enables notation for abstract syntax specified in UML, including its MOF subset (via modelElement), and supports diagram elements that notate multiple instances of metamodel elements (due to the multiplicity of modelElement). These enhancements are needed to support the case examined in Sect. 5.2. The standard BPMN DI is restricted to notating no more than one element of the BPMN metamodel.

5.2 BPMN notation on multiple metamodels

Supporting BPMN notation on both the BPMN and UML metamodels enables those interacting with systems, such as system operators, to specify or at least understand these procedures in a language suitable for them (BPMN), while system designers use more technical languages (UML) [7]. Significant inefficiency arises when these languages are not integrated. One approach to addressing this is to support BPMN notation on the UML metamodel. BPMN is the most widely used modeling standard for enterprise-level processes and provides a readily understandable notation for subject matter experts who are not computer specialists. UML is the most widely used graphical modeling standard for information systems.

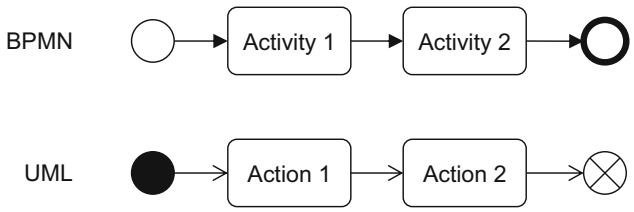


Fig. 14 BPMN process and UML activity

BPMN and UML overlap in modeling step-by-step procedures, where information and objects are passed between steps. In many cases, the differences are only in icons and line styles, with one-to-one correspondence between the underlying concepts, as in Fig. 14. Procedures in BPMN and UML are called *processes* and *activities*, respectively, while steps taken in them are called *activities* and *actions*, notated with round-cornered rectangles. Similarly, the order in which steps occur in BPMN and UML is specified by links called *sequence flows* and *control flows*, respectively. Beginnings of procedures are indicated by *start events* and *initial nodes*, while the ends are *end events* and *final nodes*.

Figure 15 shows instances of BPMN DI specializations and of BPMN and UML metamodel elements for the case of BPMN notation in Fig. 14. Figure 15 uses UML's notation for instance specifications, which are rectangles with underlined labels, where labels give the name of the BPMN DI or BPMN or UML metamodel element instantiated to the right

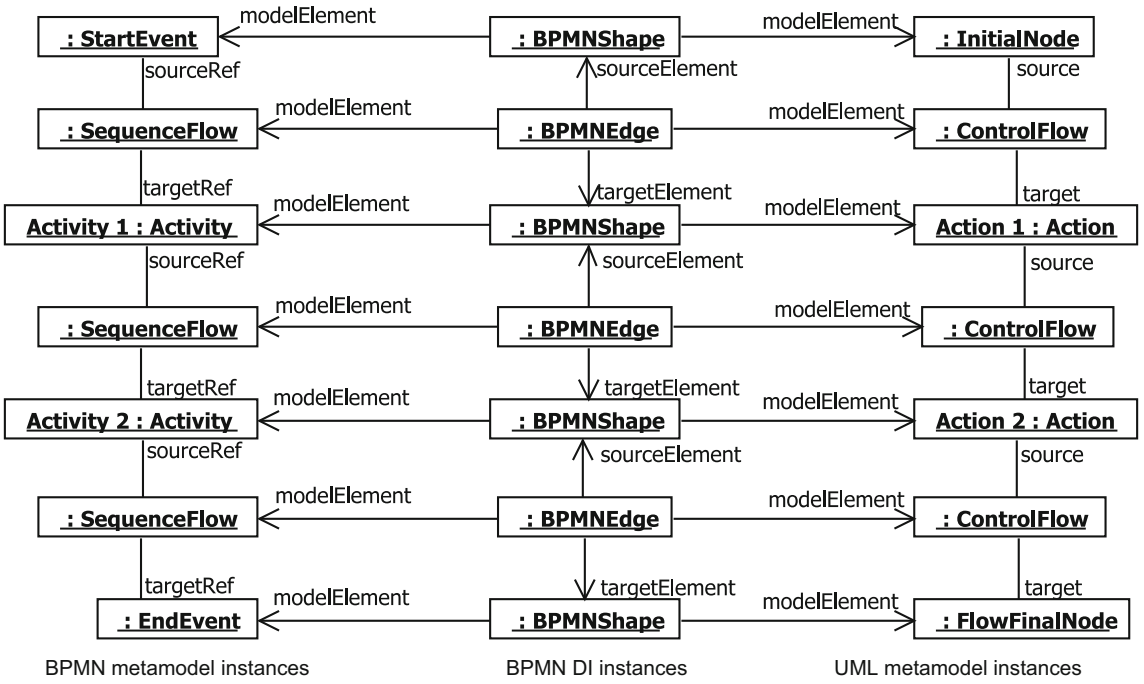


Fig. 15 Instances of BPMN DI and BPMN/UML metamodels for Fig. 14

of a colon and the instance name to the left (properties with no values and some instance names are omitted for brevity). Lines between the rectangles correspond to relationships between BPMN DI instances in the middle of Fig. 15 (see Sect. 5.1), between those and BPMN and UML metamodel instances on the sides of the figure (via modelElement), and between the BPMN and UML metamodel instances separately on each side (these are bidirectional associations, but are labeled only on one end for brevity). Text in Fig. 14 is omitted from Fig. 15 for brevity. The BPMN DI instances refer to exactly one metamodel instance each from the BPMN and UML metamodels, and vice versa, reflecting the one-to-one correspondence between underlying concepts in these cases.

Listing 9 defines a mapping used by the general DI to DG transformation in Listing 1 in Sect. 3.2 (toGraphicsTop). It operates on elements of BPMN DI (BPMNDiagramElements), dividing them into two kinds, depending on whether they identify instances of the BPMN or UML metamodels (via modelElement, see Sect. 4.1). The if statement applies toGraphics mappings to BPMN metamodel instances when they are present; otherwise, it assumes the instances are from the UML metamodel and applies toGraphicsBPMNOnUML to them. In both cases, the result of the mapping is a new DG graphical element that is added to the canvas created in Listing 1. The toGraphicsBPMNOnUML mapping in Listing 9 handles graphical elements that identify exactly one UML metamodel instance, where that element is identified by exactly one graphical element. In this case, the toGraphics mapping is called on the metamodel instances. See later listings for definitions of toGraphics and other definitions of toGraphicsBPMNOnUML.

```
mapping BPMNDiagramElement::toGraphicsTop() : GraphicalElement {
var mebpmn := self.modelElement->selectByKind(BaseElement);
if (mebpmn->isEmpty()) self.map toGraphicsBPMNOnUML();
else mebpmn->map toGraphics(self);

mapping BPMNDiagramElement::toGraphicsBPMNOnUML() : GraphicalElement
  when {self.modelElement->size() = 1 and
        self.modelElement->at(1).diagramElement->size() = 1} {
self.modelElement->map toGraphics(self); }
```

Listing 9 Instantiating DG from instances of BPMN DI

Listing 10 defines toGraphics mappings for use by Listing 9, producing instances of DG for rendering simple BPMN process diagrams, like the one at top of Fig. 14, from instances of BPMN DI like those in the middle of Fig. 15. These mappings operate on instances of the BPMN metamodel, as supplied by the first mapping in Listing 9. The first one rounds the corners of activity rectangles, while the last positions and assigns the head graphic for sequence flow arrows, see Listing 11. The second and third mapping are for event circles, with the general third mapping taking the general case, and the second adding a bold outline for end events.

```
mapping Activity::toGraphics (de:BPMNDiagramElement) : Rectangle {
de.oclAsTypeOf(BPMNShape).setRectangle (result) ;
cornerRadius := bpmnActivityCornerRadius; }

mapping Event::toGraphics (de:BPMNDiagramElement) : Circle
disjuncts EndEvent::toGraphics, BoundaryEvent::toGraphics {
de.oclAsTypeOf(BPMNShape).setCircle (result); }

mapping EndEvent::toGraphics (de:BPMNDiagramElement) : Circle
inherits Event::toGraphics {
sharedStyle := bpmnBoldOutline; }

mapping SequenceFlow::toGraphics (de:BPMNDiagramElement) : Polyline
{
de.oclAsTypeOf(BPMNEdge).setSequenceFlowPolyline(result) ; }
```

Listing 10 Instantiating DG for basic BPMN process elements via the BPMN metamodel

```
property bpmnActivityCornerRadius : PrimitiveTypes::Real = 10;

property bpmnBoldOutline = object DG::Style {strokeWidth = 5};

property bpmnArrowHead = object Marker {
  size := object Dimension {width:=10; height:=10};
  reference := object Point {x:=10; y:=5};
  member += object Polygon {
                   point += object Point{x:=0; y:=0};
                   point += reference;
                   point += object Point{x:=0; y:=10};
  sharedStyle := blackFillStyle; }; };

 query BPMNEdge::setSequenceFlowPolyline (pl : Polyline) {
   self.setPolyline (pl) ;
   pl.endMarker := bpmnArrowHead; }
```

Listing 11 Support for mappings in Listing 10

Listing 12 has the same effect as Listing 10, but goes through UML metamodel instances, as supplied by the second mapping in Listing 9, instead of BPMN. The bodies are largely the same as the corresponding mappings in Listing 10, except the UML metamodel does not have an abstraction for control nodes notated as circles, requiring separate mappings for initial and final nodes that both define BPMN event notation.

```
mapping Action::toGraphics (de:BPMNDiagramElement) : Rectangle {
de.oclAsTypeOf(BPMNShape).setRectangle (result) ;
cornerRadius := bpmnActionCornerRadius; }

mapping InitialNode::toGraphics (de:BPMNDiagramElement) : Circle {
de.oclAsTypeOf(BPMNShape).setCircle (result) ;
sharedStyle := bpmnBoldOutline; }

mapping FinalNode::toGraphics (de:BPMNDiagramElement) : Circle {
de.oclAsTypeOf(BPMNShape).setCircle (result) ;
sharedStyle := bpmnBoldOutline; }

mapping ControlFlow::toGraphics (de:BPMNDiagramElement) : Polyline {
de.oclAsTypeOf(BPMNEdge).setSequenceFlowPolyline(result) ; }
```

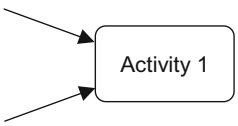
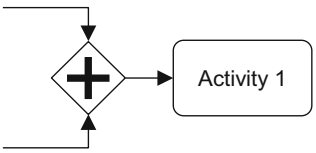
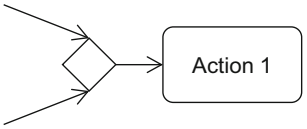
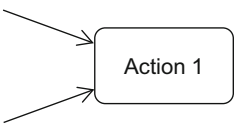
Listing 12 Instantiating DG for basic BPMN process elements via the UML metamodel

BPMN and UML have different conventions in some cases that require one-to-many or many-to-one relationships between BPMN DI and UML metamodel elements, as compared to the one-to-one relationship in Fig. 14 [7]:

- A case of one-to-many is multiple sequence flows coming into the same activity BPMN, as in the example at the upper left of Table 1. This causes the activity to execute multiple times, once for each sequence flow. A one-to-one relationship to UML would give multiple control flows into the same UML action, which causes the action to execute once, after all actions on the other ends of the control flows are finished. To use the BPMN notation on the upper left of Table 1 on the UML metamodel, the instances of UML must introduce a *merge node*, as shown in the lower left of Table 1. This causes the action to execute multiple times, once for each control flow coming into the merge node, giving the same result as BPMN.
- A case of many-to-one is *parallel gateways* before an activity, as in the BPMN example at the upper right of Table 1. This causes the activity to execute once after both sequence flows arrive at the gateway. A one-to-one relationship to UML would use a *join node* before an action, but a simpler UML model has control flows going directly into an action, as shown in the lower right of Table 1. This causes the action to execute once after both control flows have arrived, giving the same execution as BPMN.

Figures 16 and 17 show instances of BPMN DI specializations and of BPMN and UML metamodel elements for the cases on the top left and right of Table 1, respectively. In Fig. 16, the one-to-many case, the BPMN DI edges refer to exactly one metamodel instance each from the UML metamodel, while the shape refers to three. These are an action and additional merge node and control flow instances needed in UML,

Table 1 Converging flows in BPMN and UML

	Multiple executions	Single execution
BPMN		
UML		

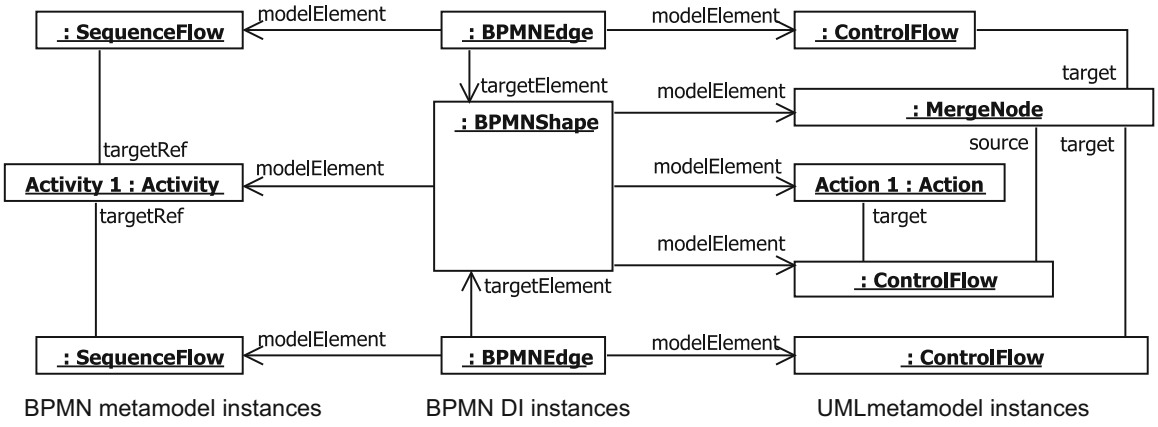


Fig. 16 Instances of BPMN DI and BPMN/UML metamodels for top left of Table 1

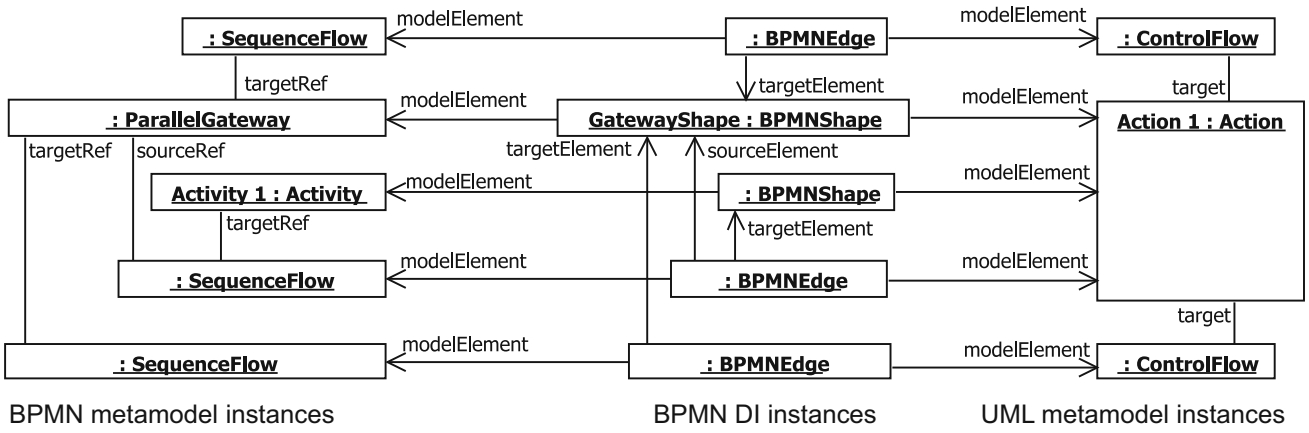


Fig. 17 Instances of BPMN DI and BPMN/UML metamodels for top right of Table 1

because the semantics of UML actions require additional elements to have the same execution as BPMN activities in this case. In Fig. 17, the many-to-one case, BPMN DI instances refer to separate instances from UML metamodel, except for the gateway and activity shapes and the sequence flow edge between them, which refer to the same UML action instance, because the action gives the same execution as those BPMN elements.

The top left of Table 1 is handled by the mappings in Listing 10, as are the similar graphics at the top right. The mapping in Listing 13 defines the rotated gateway rectangle on the top right of Table 1 (the definition of setGatewayGraphic is omitted for brevity).

```
mapping Gateway::toGraphics (de:BPMNDiagramElement) : Rectangle {
de.setGatewayGraphic (result) ; }
```

Listing 13 Instantiating DG for BPMN gateways via the BPMN metamodel

Listing 13 defines a toGraphics mapping for use by Listing 9, producing instances of DG for rendering converging BPMN sequence flows, like those on the top of Table 1, from instances of BPMN DI like those in the middle of Figs. 16 and 17. These mappings operate on instances of the BPMN metamodel, as supplied by the first mapping in Listing 9.

Listing 14 has the same effect as Listing 13 and Listing 10, but goes through UML metamodel instances, as supplied by the first mapping in Listing 9, instead of BPMN. The first two mappings in Listing 14 are for the cases of one-to-many and many-to-one relationships between BPMN diagram elements and UML metamodel instances, respectively. The first mapping focuses on the action and uses the mapping defined for those in Listing 12 to create the rounded rectangle at the

top left of Table 1. The second and third mappings detect the case of implicit join in UML using isImplicitJoin defined in Listing 15 and create the rotated gateway rectangle and arrow for the top right of Table 1.

```
mapping BPMNDiagramElement::toGraphicsBPMNOnUML() : Rectangle
  when {self.modelElement->size() > 1 and
        self.modelElement->selectByKind(Action)->size() = 1()} {
self.modelElement->selectByKind(Action)->at(1).map toGraphics(self);
}

mapping BPMNDiagramElement::toGraphicsBPMNOnUML() : Rectangle
  when {self.oclIsKindOf(BPMNShape) and
        isImplicitJoin (self)} {
return.setGatewayGraphic(self, Parallel); }

mapping BPMNDiagramElement::toGraphicsBPMNOnUML() : Line
  when {self.oclIsKindOf(BPMNEdge) and
        isImplicitJoin (self)} {
self.oclAsTypeOf(BPMNEdge).setSequenceFlowPolyline(result) ; }
```

Listing 14 Instantiating DG for BPMN gateways via the UML metamodel

```
query isImplicitJoin (de : BPMNDiagramElement) : Boolean {
  var result : Boolean = false;
  var me := de.modelElement;
  if (me->size() = 1 and me.diagramElement->size() = 3)
    { var me1de = me.diagramElement;
      if (me1de->selectByKind(BPMNEdge)->size()=1 and
          me1de->selectByKind(BPMNShape)->size()=2)
        { var me1dee = me1de->selectByKind(BPMNEdge).at(1);
          var me1des1 = me1de->selectByKind(BPMNShape).at(1);
          var me1des2 = me1de->selectByKind(BPMNShape).at(2);
          if (me1de->includes(me1dee.source) and
              me1de->includes(me1dee.target))
            result = true; }; };
  return result;}
```

Listing 15 Support for mappings in Listing 14

BPMN has some simplifying constructs for procedures that are more cumbersome to express in UML. An example of this is termination of activities due to occurrences of events, as shown in Fig. 18. Circles in BPMN on the boundaries of activity rectangles notate events occurring while a BPMN activity is going on, in this case a timer event on the top and general change event on the bottom. Similarly, in UML the hour glass shape and lower rounded rectangle notate the same kind of events (UML has predefined actions for detecting events). If the timer or change event occurs during the activity or action, the activity or action is terminated and the procedure continues along the sequence or control flow going out of the event that occurred. This behavior is built into the BPMN concept of *boundary events*, providing a more compact notation than UML, which introduces *interruptible regions* containing the actions and events involved, notated with a dashed border. In the UML portion of Fig. 18, when the central action is initiated by its incoming control flow, the event elements around it in the region are activated as well. Whichever of these finishes first, either because one of the events occurs or the central action finishes, terminates the others occurring in the region, and the procedure continues along the control flow going out of the event that occured or

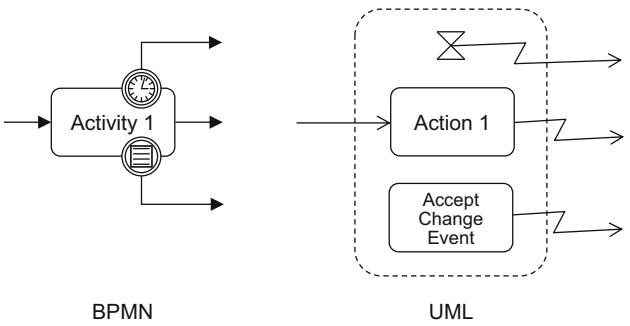


Fig. 18 BPMN boundary events and UML interruptible region

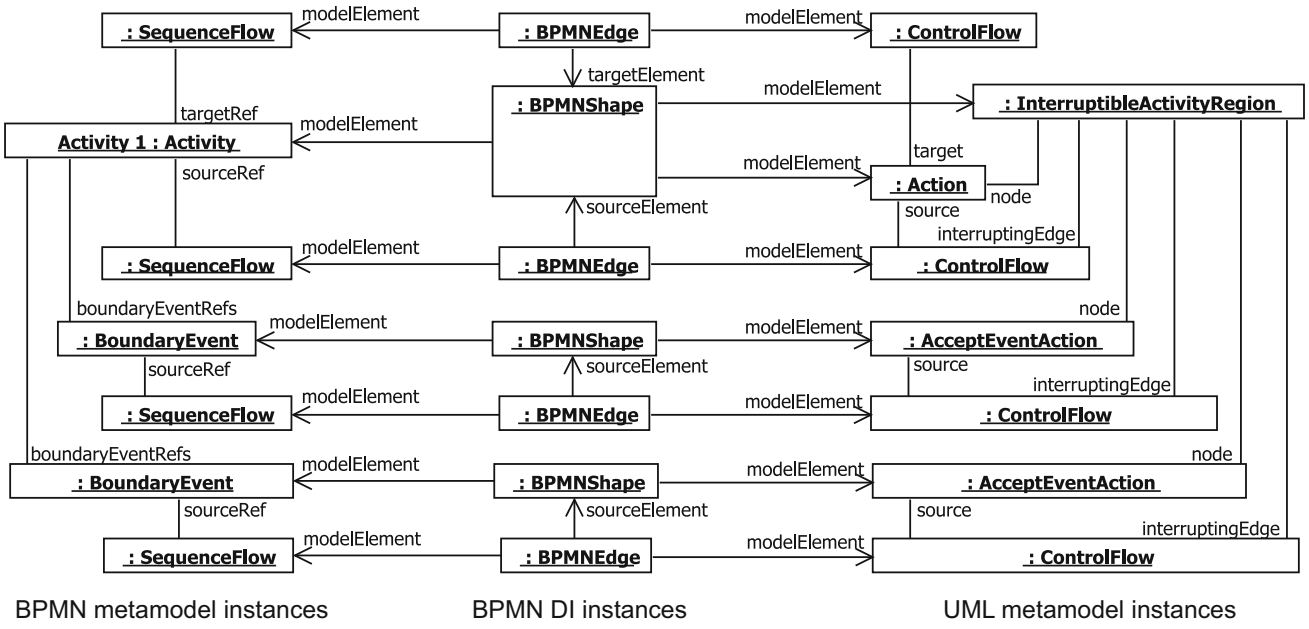


Fig. 19 Instances of BPMN DI and BPMN/UML metamodels for Fig. 18

the central action. These flows are notated with zigzag lines to indicate they terminate the other elements in the region when flow crosses the boundary.

Figure 19 shows instances of BPMN DI specializations and of BPMN and UML metamodel elements for the case in Fig. 18. The BPMN DI instances refer to exactly one metamodel instance each from the BPMN and UML metamodels, except for the activity, which refers additionally to the interruptible region. This follows the pattern of Fig. 16 to provide the more compact BPMN notation for UML's metamodel instances.

Listing 16 defines toGraphics mappings for use by Listing 9, producing instances of DG for rendering boundary events, like those on left of Fig. 18 from instances of BPMN DI like those in the middle of Fig. 19. The mapping operates on instances of the BPMN metamodel, as supplied by the first mapping in Listing 9. It uses setTimerEventGraphic and setConditionalEventGraphic to produce BPMN symbols for timers and conditionals inside the event circles (definitions of these are omitted for brevity). Listing 17 has the same effect as Listing 16, but goes through UML metamodel instances, as supplied by the first mapping in Listing 9, instead of BPMN.

```
mapping BoundaryEvent::toGraphics (de:BPMNDiagramElement) : Group
  inherits Event::toGraphics {
var ed := self.eventDefinitionRefs->union(self.eventDefinitions);
if (ed->selectByKind(TimerEventDefinition)->notEmpty())
  de.setTimerEventGraphic(result);
elif (ed->selectByKind(ConditionalEventDefinition)->notEmpty())
  de.setConditionalEventGraphic(result); }
```

Listing 16 Instantiating DG for BPMN boundary events via the BPMN metamodel

```
mapping AcceptEventAction::toGraphics (de:BPMNDiagramElement) :
Group {
var ev := self.trigger.event;
de.setCircle (result);
if (ed->selectByKind(TimeEvent)->notEmpty())
  de.setTimerEventGraphic(result);
elif (ed->selectByKind(ChangeEvent)->notEmpty())
  de.setConditionalEventGraphic(result); }
```

Listing 17 Instantiating DG for BPMN boundary events via the UML metamodel

6 Conclusion

This paper demonstrates concrete graphical syntax (notation) in many-to-many relationships with abstract syntax (metamodels), giving definers of graphical languages a way to address the challenges of notational variety for the same or similar underlying language concepts, as described in Sect. 1. Despite the history and benefits of separately modeling abstract and concrete syntax, many-to-many relationships between them have not been described previously, as shown in the review of related work in Sect. 2, the more common application being interchange of visual information between graphical modeling tools, such as positions of nodes and routings of lines. To ensure the demonstrations are useful to definers of graphical languages, the paper employs an open standard for specifying graphical syntax (DD) and examines cases drawn from widely used graphical modeling languages (UML and BPMN), in Sects. 3, 4 and 5. These cases demonstrate multiple notations for the same metamodel and the same notation for multiple metamodels. From these, we conclude that it is feasible for definers of graphical languages to specify notational views of the same metamodel adapted for particular applications and the same notational view for related metamodels, improving communication efficiency among users of the languages and their tools.

Examining these cases also identified topics for future work, in particular, tradeoffs arising from how differently to construct concrete and abstract syntax for the same language, avoiding overspecification of concrete syntax, and issues in modeling lines that have no objects at their ends. Another future topic is reuse, for example, reusing DG mappings to the same concrete syntax across multiple abstract syntaxes, and design patterns for DD.

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison Wesley, Boston (2006)
2. Bock, C.: UML without pictures. In: IEEE Software Special Issue on Model-Driven Development 20(5):33–35 (2003)
3. Object Management Group: Diagram Definition. http://omg.org/spec/DD (2014). Accessed 9 April 2016
4. Elaasar, M., Labiche, Y.: Diagram definition: a case study with the UML class diagram. In: Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science 6981, pp. 364–378 (2011)
5. Object Management Group: OMG Unified Modeling Language, version 2.5. http://www.omg.org/spec/UML/2.5 (2015). Accessed 9 April 2016
6. Object Management Group: Business Process Model and Notation, version 2.0". http://www.omg.org/spec/BPMN/2.0 (2013). Accessed 9 April 2016
7. Bock, C., Barbau, R., Narayanana, A.: BPMN profile for operational requirements. J. Object Orient. Technol. **13**(2), 1–35 (2014)
8. Object Management Group: Action Language for Foundational UML. http://www.omg.org/spec/ALF (2013). Accessed 9 April 2016
9. Schenck, D., Wilson, P.: Information Modeling the EXPRESS Way. Oxford University Press, Oxford (1994)
10. W3C OWL Working Group: OWL 2 Web Ontology Language Document Overview. http://www.w3.org/TR/owl2-overview (2012). Accessed 9 April 2016
11. Rath, I., Okros, A., Varro, D.: Synchronization of abstract and concrete syntax in domain-specific modeling languages. Softw. Syst. Model. **9**(4), 453–471 (2010)
12. Krahn, H., Rumpe, B., Volkel, S.: Integrated definition of abstract and concrete syntax for textual languages. In: Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science 4735, pp. 286–300 (2007)
13. Baar, T.: Correctly defined concrete syntax. Softw. Syst. Model. **7**(4), 383–398 (2008)
14. Wile, D.: Abstract syntax from concrete syntax. In: Proceedings of the 19th International Conference on Software Engineering, pp. 472–480 (1997)
15. Muller, P., Fondement, F., Fleurey, F., Hassenforder, M., Schneckenburger, R., Gerard, S., Jezequel, J.: Model-driven analysis and synthesis of textual concrete syntax. Softw. Syst. Model. **7**(4), 423–441 (2008)
16. Poruban, J., Forgac, M., Sabo, M.: Annotation based parser generator. Compt. Sci. Inf. Syst. **7**(2), 291–307 (2010)
17. Alone, Y., Deshmukh, V.: Annotation based innovative Parser Generator. Int. J. Adv. Res. Comput. Sci. Manag. Stud. **2**(1), 498–501 (2014)
18. Genesereth, M., Nilsson, N.: Logical Foundations of Artificial Intelligence. Morgan Kaufman, Los Altos (1987)
19. Flanagan, D.: JavaScript: The Definitive Guide. O'Reilly Media, Sebastopol (2011)
20. Backus, J., Bauer, F., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A., Rutishauser, H., Samuelson, K., Vauquois, B., Wegstein, J., van Wijngaarden, A., Woodger, M.: Revised report on the algorithm language ALGOL 60. Commun. ACM **6**(1), 1–17 (1963)
21. International Standards Organization: Information Technology—Syntactic Metalanguage—Extended BNF. http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip (1966). Accessed 9 April 2016
22. Tennent, R.: The denotational semantics of programming languages. Commun. ACM **19**(8), 437–453 (1976)
23. Backlund, B., Hagsand, O., Pehrson, B.: Generation of visual language-oriented design environments. J. Vis. Lang. Comput. **1**(4), 333–354 (1990)
24. Arefi, F., Hughes, C.E., Workman, D.A.: Automatically generating visual syntax-directed editors. Commun. ACM **33**(3), 349–360 (1990)
25. Flatscher, R.: Metamodeling in EIA/CDIF—meta-metamodel and metamodels. ACM Trans. Model. Comput Simul. **12**(4), 322–342 (2002)
26. Object Management Group: Reference Metamodel for the EXPRESS Information Modeling Language. http://www.omg.org/spec/EXPRESS (2015). Accessed 9 April 2016
27. W3C: Extensible Markup Language. http://www.w3.org/TR/xml11 (2006). Accessed 9 April 2016
28. Object Management Group: Meta Object Facility. http://omg.org/spec/MOF (2015). Accessed 9 April 2016
29. Object Management Group: XML Metadata Interchange. http://omg.org/spec/XMI (2015). Accessed 9 April 2016
30. The Eclipse Foundation: Graphical Modeling Framework. http://www.eclipse.org/modeling/gmp (2013). Accessed 9 April 2016
31. The Eclipse Foundation: Model Development Tools. http://www.eclipse.org/modeling/mdt/?project=uml2 (2015). Accessed 9 April 2016

32. Object Management Group: Diagram Interchange. http://doc.omg.org/formal/06-04-04 (2006). Accessed 9 April 2016
33. Object Management Group: Systems Modeling Language, Version 1.4. http://www.omg.org/spec/SysML(2015). Accessed 9 April 2016
34. Object Management Group: Interaction Flow Modeling Language. http://www.omg.org/spec/IFML (2015). Accessed 9 April 2016
35. Object Management Group: Semantics of a Foundational Subset for Executable UML Models. http://www.omg.org/spec/FUML (2013). Accessed 9 April 2016
36. Schildt, H.: Java: The Complete Reference. McGraw-Hill, New York (2014)
37. W3C: XML Schema. http://www.w3.org/XML/Schema (2012). Accessed 9 April 2016
38. Object Management Group: UML Profile for BPMN 2 Processes. http://www.omg.org/spec/BPMNProfile (2014). Accessed 9 April 2016
39. Krasner, G., Pope, S.: A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. J. Object Orient. Prog. **1**(3), 26–49 (1988)
40. Grundy, J., Hosking, J.: The MViews framework for constructing multi-view editing environments. N. Z. J. Comput. **4**(2), 31–40 (1993)
41. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional, Boston (1994)
42. W3C: Scalable Vector Graphics. http://www.w3.org/TR/SVG11 (2011). Accessed 9 April 2016
43. Object Management Group: Query/View/Transformation Specification. http://www.omg.org/spec/QVT (2015). Accessed 9 April 2016

Conrad Bock is a Computer Scientist at the U.S. National Institute of Standards and Technology's Engineering Laboratory, specializing in formal system, product, and process modeling. He was the founding editor for Activity and Action modeling in the Unified Modeling Language (UML) and Systems Modeling Language at the Object Management Group (OMG), as well as a primary contributor to interaction modeling in the Business Process Model and Notation (BPMN) and to Diagram Definition (DD). Conrad was lead developer of the UML Profile of BPMN Processes and of logical formalization for UML in the Foundational Subset for Executable UML Models. He is lead author of over 25 journal articles and book chapters. He is currently co-chair of the DD Revision Task Force at OMG and project leader for integration of systems and engineering analysis models at NIST.

Maged Elaasar is a senior software architect at the Jet Propulsion Laboratory (Caltech/NASA), where he leads R&D projects in model-based systems engineering. Prior to that, he was a senior software architect at IBM, where he led R&D projects in model-driven engineering. He holds a Ph.D. in Electrical and Computer Engineering and M.Sc. in Computer Science from Carleton University (2012, 2003), and a B.Sc. in Computer Science from American University in Cairo (1996). He has received 12 U.S. patents and authored over 20 peer-reviewed journal and conference articles. He was a primary contributor to Diagram Definition (DD) at the Object Management Group and is currently a co-chair of the DD Revision Task Force. Maged is also the founder of Modelware Solutions, a software consultancy and training company with international clients and affiliations to international laboratories, such as SQUALL (Canada), CEA LIST (France), SnT (Luxembourg), and Simula (Norway).