Systematic Generation of Standard Compliant Tool Support of Diagrammatic Modeling Languages

Alexis Fouché, Florian Noyrit, Sébastien Gérard
CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, P.C. 174, Gif-sur-Yvette, 91191, France
alexis.fouche@cea.fr, florian.noyrit@cea.fr, sebastien.gerard@cea.fr

Maged Elaasar
Department of Systems and Computer Engineering
Carleton University, Ottawa, Canada
Crossplatform Software Inc.
Ottawa, Canada
melaasar@gmail.com

Abstract—In the Model-Driven Engineering community, the abstract syntax of modeling languages is usually defined and implemented using metamodeling techniques. However, it is not the case for the concrete syntax of graphical modeling languages. Indeed, this concern is mostly specified by informal means. This practice leaves considerable leeway in the implementation and raises several standards compliance issues. Hence, toolsmiths can only rely on their interpretation of the standard and lack of systematic way to build conforming tool support. In this context, a first normative specification of the concrete syntax of UML 2.5 has been recently released using Diagram Definition. In this paper, we propose an approach that uses those formal specifications to systematically generate modeling language tool support that guarantees compliance to standard notation. We assess the approach on a subset of the UML class diagram implemented within the open-source Papyrus tool.

Index Terms—Diagrammatic languages, standard-compliance, tooling support, MDE, UML.

I. INTRODUCTION

The Model-Driven Engineering community promotes the systematic use of models throughout the development of software and systems. Metamodeling techniques such as MOF [1] or EMF [2] are commonly used to define the abstract syntax that models must conform with. Those techniques have been used to standardize modeling languages and thereby enable consistent manipulation and interpretation of models. The serialization of models has also been standardized with XMI [3] to enable model interchange. Finally, many tools and methods have been developed to exploit metamodeling techniques. This rich ecosystem of standards, methods and tools foster the development of modeling languages required to address various engineering domains and concerns.

Several modeling languages adopt a diagrammatic concrete syntax. Such a graphical notation, when designed with care, greatly help designers in expressing and analyzing their models [4]. The definition of this concrete syntax should complement the abstract syntax of a modeling language. However, in practice, the definition of the concrete syntax and its mapping with the abstract syntax is unfortunately often specified with “by-example” definitions and natural-language descriptions in the specifications.

This practice prevents, or at least makes difficult, any implementation of a graphical language from being strictly standard compliant by leaving room for interpretation. Moreover, it compromises diagram interchange support between tools, as two implementations of the same standard language will most probably be different from each other and could even be conflicting. Even though several frameworks can be used to facilitate the implementation of diagrammatic languages, toolsmiths willing to implement a diagrammatic language lack of systematic means to build the concrete syntax specification and basically do it from scratch.

To address this flaw, the Diagram Definition standard (DD) [5] has been issued by the OMG for formally defining diagrams, facilitating the interchange and ensuring consistent rendering of diagrams between conforming tools. This standard has been recently used to specify the UML 2.5 notation ([6] Annex B).

In this paper, we do not develop a proposal for a new language specification technique. Several tools provide such techniques and have already proved their worth, such as Sirius [7], GMF-Tooling [8] or MetaEdit+ [9]. Rather, this paper aims at assessing the Diagram Definition potential as a standard visual modeling language specification framework in a practical context.

After going through the basis of a diagrammatic language implementation in section 2 and a brief overview of the DD standard in section 3, we assess in section 4 how a DD-based specification of a graphical modeling language notation can be adapted for automatic tool support generation. The approach, being systematic, reduces considerably the development effort and more importantly guarantees a standard-compliant implementation of diagrammatic notation. We report in section 5 on the prototyping of the approach on a subset of the UML Class diagram, and bring further discussion in section 6.

II. DEFINING AND IMPLEMENTING A DIAGRAMMATIC MODELING LANGUAGE

In [4], the author states that the concrete syntax of a diagrammatic modeling language consists of the visual vocabulary (e.g., lines, areas and textual elements such as label) and a set of compositional rules (or visual grammar).
The semantic constructs (also known as the abstract syntax) is typically defined by a metamodel. Graphical symbols are then mapped to the semantic constructs they represent to give them meaning.

Therefore, to follow a model-driven engineering approach, the definition of diagrammatic modeling language can consist of:

- The metamodel of the abstract syntax (e.g. the UML metamodel) denoted with MOF, which will capture the semantic constructs of the modeling language.
- The metamodel of the concrete syntax which will capture the visual grammar of the modeling language.
- The mapping from the concrete syntax to the abstract syntax, which will assign meaning to the visual grammar using cross-references.
- The mapping from the visual grammar to the visual vocabulary (graphical symbols) captured using a model-to-model (M2M) transformation to a graphics language.

Several approaches have been proposed for the implementation of diagrammatic modeling languages: Grammar-based approaches such as eXtended Positional Grammars [10] or Layered Graph Grammars [11], Metamodel-based approaches like TIGER [12], GMF-Tooling, Sirius and many other frameworks such as MetaEdit+, AToM³ [13] or constrain-based layout definition as proposed in [14]. Yet, none of them considered standard compliance (i.e., diagrams look and behave exactly the same in different tools) and interoperability requirements i.e., diagrams can be exchanged between tools.

For the tooling support of the abstract syntax, a toolsmith can directly rely on a standard MOF-based metamodel provided by the language designer to build the implementation (for instance using EMF or the like metamodeling framework). However, whatever the implementation framework used, in practice, the tooling support for the concrete syntax is manually implemented through a cumbersome process that heavily relies on the toolsmith’s designer expertise. Ultimately the implementation will be based on their interpretation of the standard.

The problem we address in this paper is how to systematically build standard-compliant diagrammatic language tooling support.

III. GRAPHICAL NOTATION FORMALIZATION WITH DIAGRAM DEFINITION

Before presenting our approach, we give a short introduction to the OMG’s Diagram Definition standard and a brief overview of the notation standardization effort for UML 2.5.

A. Diagram Definition (DD)

DD specifies a framework for “modeling and interchanging graphical notation” [5]. To this end, DD provides an architecture that distinguishes two kinds of graphical information:

- The information that users can control and which is encoded for interchange between tools (e.g., the layouts and notational options). This is captured thanks to the Diagram Interchange (DI) metamodel (illustrated in Fig. 1).
- The information that users do not control, which are not meant to be interchanged and that must be the same across all conforming tools (e.g., normative shapes and line styles). This information is captured using the Diagram Graphics (DG) metamodel. Fig. 2 depicts some of the visual vocabulary available in DG.

Using DD standard, a diagrammatic language specification should include the following artifacts:

- A specialization of the DI metamodel, which is used to specify the visual grammar of the notation.
- For each element of the DI specialization metamodel, references to the corresponding elements in the language’s abstract syntax model which maps the concrete syntax to the semantic constructs.
- A mapping from the DI specialization metamodel to the DG metamodel. It basically captures the mapping from
the graphical grammar to the visual vocabulary. This mapping can be expressed with QVT [15].

This architecture is aligned with the generic schema proposed in II to define a graphical modeling language. Fig. 3 denotes how this architecture is instantiated for UML.

In this example, diagrams produced with a conforming tool to these requirements will consist of: a UML model, which contains instances of the UML semantic constructs, namely the model elements (A in Fig. 3), a notation model conforming to UMLDI (a specialization of the DI metamodel for UML), containing instances of the visual grammar (B in Fig. 3). This model also provides a mapping between the semantic and notational model elements, represented in section C in Fig. 3. In addition, an instance of the DG model can be generated using the UMLDI→DG mapping to render the graphical representation with appropriate graphical vocabulary. It is obtained by applying the model transformation to the DI specialization model. This is represented in section D in Fig. 3.

B. Standardization of the UML Notation

Until the release of UML 2.4.1, the notation of the language was described informally, mainly relying on a set of examples [16]. As part of its continuous effort to improve the standardization of the UML, the OMG has issued a normative annex B to the UML 2.5 to define its concrete syntax using DD. This annex contains UMLDI, which is a refinement of the DI metamodel for the UML notation.

As illustrated in Fig. 4, the UMLDI metamodel introduces constructs specific to the notation of UML. It notably adds constructs for labels, compartments, compartmentable shapes, stereotype labels and edge labels. To this end, DI::Shape is specialized by the UMLCompartmentableShape and UMLLabel elements. The UMLCompartment element specializes DI::DiagramElement. The UMLEdge element specializes DI::Edge and UMLDiagramElement. On this basis, different associations are specified between UMLDI elements that refine corresponding associations in DI through redefinition or subsetting constraints placed on their association end properties. For example, the compositions from UMLCompartmentableShape to UMLCompartment and from UMLCompartment to UMLDiagramElement refine the self composition on DI::DiagramElement by subsetting its end properties: owningElement and ownedElement. Also, UMLEdge's source and target properties (and their opposite association ends) are specified to redefine DI::Edge's source and target properties (and their opposite association ends).

Moreover, UMLDI defines the mapping between diagram elements and elements of the UML abstract syntax by redefining the modelElement property of DI::DiagramElement. For instance, in Fig. 5, the graphical elements UMLClassifierShape, UMLMultiplicityLabel, UMLStateShape, and UMLActivityDiagram are mapped respectively to the abstract syntax elements UML::Classification::Classifier, UML::CommonStructure::MultiplicityElement, UML::StateMachine::State, and UML::Activities::Activity. In addition, a graphical construct might not be mapped to any element of the abstract syntax, as it is the case for UMLCompartment.

Considering these informational aspects provided by the UMLDI metamodel together with the UML metamodel itself, the schema proposed in section II to define graphical modeling languages seems to be fully covered with the exception of the mapping from the concrete syntax to the graphical vocabulary. Indeed, the QVT mapping from UMLDI to DG is currently not provided by the UML specification, but is subject of an ongoing work and planned to be released in a later version.
IV. DD-BASED SYSTEMATIC GENERATION OF TOOL SUPPORT

We present in this section an approach that evaluates the potential of the Diagram Definition standard as a formal specification of a graphical language’s concrete syntax to automatically generate graphical editing tool support. Indeed, the DD specification together with the MOF-based language’s abstract syntax specification constitute the complete formal specification of the language and thereby should be a good starting point for a toolsmith to drive the model-based generative process of editing tools.

A Diagram Definition based specification needs to have certain kinds of information in order to be usable to generate editing tooling. Firstly, the visual grammar, or the specialized DI metamodel, provided by the specification must be complete (i.e., fully mapped to the abstract syntax that needs to be supported) and fully constrained (i.e., it fully refines the basic building blocks in DI) to formally reflect the targeted notation. Secondly, the specialized DI metamodel must provide a vocabulary of diagram elements corresponding to a set of graphical abstractions (e.g., shape, edge, label and compartment) with well-defined editing behavior. Thirdly, the mapping with the abstract syntax needs to be unambiguous such that we can automatically and correctly interpret editing requests. Finally, the mapping to DG must be complete and translatable to the Model-View-Controller (MVC) pattern commonly used by modern graphical editing frameworks.

The UML class diagram may be the most wildly used and most familiar diagram example. Therefore the evaluation approach presented here is mainly driven by this example. This choice is consistent with the use case of the implementation presented in section V.

A. Capturing the Visual Grammar

The concrete syntax specification provided by the DD-based UMLDI metamodel does provide a set of identification and composition mechanisms allowing interpreting the whole structure of a UML class diagram. However, these are too general and do not provide the complete set of constraints required to formally verify the well-formation of a diagram.

For instance, Fig. 6 shows how the compartment element is specified in UMLDI. We can see that a compartmentalizable shape can compose any number of compartments. This is inherited by a classifier shape, which specializes compartmentalizable shape, as shown in Fig. 7. This mechanism is sufficient enough to interpret for instance the composition of a class representation as a classifier shape associated to three compartments. However, it does not add constraints that classifier shapes can only compose certain kinds of compartments per the UML specification. Fig. 8 illustrates this issue with an instance of classifier shape (for a Class Person) that conforms to UMLDI but not to UML specification (a Class shape cannot own an enumeration literal compartment).
These additional classes and associations follow the same refinement mechanism used in UMLDI. Notably, the introduced attributes are redefining the \texttt{DI::DiagramElement} attributes: \texttt{ownedElement} and \texttt{owneringElement}.

![Diagram of graphical syntax for the class shape in UML](image)

**Fig. 9.** Subset of the graphical syntax for the class shape in UML

### B. Capturing the Graphical Elements’ Behavior

A model-driven framework for building diagram editors (e.g., the Graphical Modeling Framework [17] on Eclipse) often follows a MVC architecture. Furthermore, it usually comes with a set of prepackaged controllers designed for the most generic diagram abstractions (e.g., shape, edge, label and compartment). These controllers will handle user interactions (e.g., drag & drop and link redirection) and compute the impact on the model accordingly based on the specified mapping between the concrete syntax and the abstract syntax. Depending on the chosen framework, this set of prepackaged diagram abstractions may vary, as well as the mechanism for designing customized controllers.

The Diagram Interchange (DI) metamodel, which is a generic tooling framework can rely on for identifying diagram abstractions, makes the distinction between three types of diagram abstractions only: diagrams, shapes and edges. Even if we consider a simplified UML class diagram, this DI classification is too high level to accurately map to the expected kinds of controllers. In practice, we show in section V.B that this flaw can be easily addressed using an augmenting classification mechanism, such as a UML profile, to provide the missing graphical constructs’ behavior identification.

### C. Interpreting the Mapping with the Abstract Syntax

For the sake of simplicity, the approach presented in this paper only considers one-to-one mappings between concrete syntax and abstract syntax elements. However, generating tool support for a graphical modeling language still requires to automatically and unambiguously map each graphical relation to the corresponding association in the abstract syntax model. This process is not always deterministic as there is no specific mapping provided between the UMLDI associations and the UML associations.

**TABLE I.** The different containers of a UML::Constraint in UML and the corresponding composite associations

<table>
<thead>
<tr>
<th>Associations Analysis</th>
<th>Element</th>
<th>Associated element</th>
<th>Composite Associations</th>
<th>Ambiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UML::Namespace</td>
<td>ownedRule</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UML::StateInvariant</td>
<td>invariant</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UML::ParameterSet</td>
<td>condition</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UML::Extend</td>
<td>condition</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UML::Action</td>
<td>localPrecondition</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>localPostcondition</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the course of experimenting with this problem, an (effectively simple mapping) algorithm was developed to introspect the UML metamodel and compute the possible UML associations for every pair of UML elements. Basically, for every class X (e.g., Class, Package and Interface) in the UML metamodel, the algorithm lists every other class Y that can be associated to X. Every match is considered as an association case. If more than one association is found for an association case, it is considered as an ambiguity. For instance, **TABLE I.** consists of five association cases, amongst which one is ambiguous.

Because of the existence of extensive generalization hierarchies in the UML metamodel, the number of association cases – ambiguous or not – is high. For instance, the five association cases listed in **TABLE I.** will be duplicated for every class specializing **UML::Constraint**. Fig. 10 together with **TABLE II.** provide better insight on the issue. **TABLE II.** shows the result of an association analysis on the example metamodel of Fig. 10 consisting of five classes, three generalizations and four associations. The result lists six association cases, amongst which five are ambiguous. It is clear that without the generalizations the containment analysis result would be simpler and list only four unambiguous association cases – one for each association. To prevent the results from being polluted, the algorithm ignores duplicated ambiguous cases. For instance, considering the metamodel shown in Fig. 10, the [C, D] case where C is associated to D is considered as a duplicate of both the [A, D] case (as C is a specialization of A) and the [C, B] case (as D is a specialization of B). Eventually the algorithm will list one unambiguous case ([A, B]) and two ambiguous cases ([C, B] and [A, D]) for the whole metamodel. The remaining cases are considered as duplicates.

This ambiguity issue can be at least partially tackled by the abstract syntax metamodel itself. Indeed, the MOF redefinition mechanism allows an association end to redefine an inherited association end. In such situations, an automated process can select which association end should be used in a deterministic manner. For instance, in the metamodel shown in Fig. 10, if the association end of R3 redefines the association end of R1, then R3’s end should always be used in place of R1’s end whenever an instance of A is graphically related to an instance of D. Hence the association case where A is associated to D would be disambiguated.

352
When building an abstract syntax metamodel, a language designer should always use this method whenever possible, i.e., when one association specializes another. For example, the UML metamodel already applies this method in the relevant cases. Unfortunately this method does not resolve the situation denoted in TABLE I : specifying association end localPrecondition as a redefinition of association end localPostcondition would not make sense as the former’s association does not specialize the latter’s. The other metamodeling method to resolve this issue is to avoid using generalizations in the abstract syntax that lead to such ambiguities. However, this cannot be done without careful analysis to avoid unnecessarily increasing the size of the metamodel and with it the size and complexity of any related transformation or mapping.

Eventually, the remaining ambiguous cases cannot be resolved automatically and require either the language designer intervention (by specifying explicitly a reverse mapping from concrete to abstract syntax), or the user interaction every time an ambiguity arises. If we run the algorithm over the whole UML metamodel as provided by the EMF UML2 implementation on Eclipse and considering only composition associations, we get the following results: UML includes 172 composition cases, 34 of which are ambiguous. Moreover, 5 of these ambiguous cases can be reduced if we consider the redefinition mechanism, but only one is completely disambiguated; thus leaving 33 unsolved ambiguous composition cases for the whole UML abstract syntax. As an example, always considering only composite associations, a UML::LoopNode element can reference a UML::InputPin element using one of these three attributes: loopVariableInput, structuredNodeInput, or node. The UML::LoopNode::loopVariableInput attribute is a redefinition of the UML::StructuredActivityNode::structuredNodeInput attribute (UML::LoopNode being a specialization of UML::StructuredActivityNode), thus reducing the ambiguity as it can be automatically determined that a UML::LoopNode should never reference a UML::InputPin as a UML::StructuredActivityNode::structuredNodeInput but rather as a UML::LoopNode::loopVariableInput. However this is not enough to completely disambiguate the association case as the UML::StructuredActivityNode::node attribute also remain a valid option.

D. Specifying the Graphical Vocabulary

The DG metamodel provides a generic graphical vocabulary that is used to visually render concrete syntax elements. The process used to produce a DG instance model from a DI specialization instance model is captured by the DI→DG mapping.

When the user uses the model editor to create and manipulate diagrams, he directly interacts with the rendered view of the diagram. Hence the editor must clearly identify the graphical boundaries of every element rendered on the diagram. In order to handle these graphical interactions through an MVC pattern, they must be forwarded correctly to the DI specialization instance model.

To meet these requirements, the DI→DG mapping must first provides traceability between DI specialization elements and DG elements, as well as a way to identify the effective graphical boundaries of a DG element. Furthermore, the diagram editor must be responsive and display any modifications made to the model in real time. To this end, performing the transformation on the whole DI specialization instance model every time a modification arises clearly is a cumbersome solution. Thus the DI→DG mapping should be modular and allow re-applying the transformation only on a subset (i.e. incremental transformation) of the DI specialization model.

V. VALIDATION OF THE APPROACH

A. Usecase

In order to evaluate the pros and cons of the Diagram Definition specification as a driver for generating graphical editing tooling, the approach was implemented in a practical context: generating a simplified UML class diagram editor from UML DD. The simplified UML class diagram that was considered consists of the following three elements: UML::Class, UML::Package, and UML::Association, including all specific labels and compartments for both Class and Package.

Any framework providing a model-driven approach for building diagram editors should be valid as the target platform. In our context, we chose to prototype the approach in the Eclipse environment [18] using the GMF-Tooling framework and the Papyrus [19] UML model editor.

B. Implementation

The implementation adopts an automated generative process, taking a DI specialization metamodel (e.g., UMLDI) and its corresponding abstract syntax metamodel as an input,
and producing a custom Papyrus model editor as an output. The implementation uses the platform-specific concrete syntax metamodel of GMF-Tooling, (the GMFGen metamodel) as an intermediary model, as it will be explained later in this section.

The MDE process that we propose in order to evaluate the use of DD specifications for generating model editors is illustrated in Fig. 11.

It splits up in three phases:

1. The DI specialization metamodel is parsed; all the possible compositions are listed and mapped with the compositions of the abstract syntax. (Notice that we limited the analysis to compositions in the context of our prototype only, without limiting the approach in general)
2. An intermediary GMFGen model is populated with information from the input models as well as the computed information in step 1.
3. The regular GMF generation process is invoked on the GMFGen model to generate the Papyrus model editor as an Eclipse plugin.

Eventually, the generated model editor would rely on the DI→DG transformation at runtime to graphically render the diagram. The implementation presented here does not include such support, but directly relies on the existing GMF-Tooling mechanism.

GMF-Tooling provides support for graphically rendering a diagram through a set of Java classes called FigureViewMap. Each of these classes implements one element of the graphical vocabulary. A default set of FigureViewMap classes is provided to support some of the most commonly used graphical vocabulary elements. GMF-Tooling also provides means to implement customized FigureViewMap classes to handle specific graphical vocabulary elements. The FigureViewMap elements are working in a hierarchical manner, i.e. the rendering of a FigureViewMap element calls the rendering of all of its children. The global architecture of this FigureViewMap mechanism seems to be a good basis for implementing a DI→DG mapping support. Providing a FigureViewMap element access to the corresponding DI specialization model element could be done without major difficulties. Then the FigureViewMap element could monitor its associated DI specialization model element and perform the rendering whenever a modification arises. Thus it would trigger the DI→DG transformation on its associated DI element only, as well as the rendering of all its children. This solution raises some consistency issues and would require an impact analysis to guarantee the overall steadiness of the rendering of the diagram. But it is a promising lead for implementing the support of the rendering based on the DI→DG mapping.

The GMF-Tooling framework provides support for a set of default graphical constructs such as nodes, edges, labels or compartments. The support consists of controllers that render the graphical constructs corresponding to the concrete syntax elements and that handle the user interactions with the diagram, namely: creation, deletion, drag & drop, edge redirection, text label edition, etc. Such support brings a certain amount of complexity: the graphical constructs must not only behave accordingly to the language specification, but the right modifications must also be applied to the abstract syntax model. As a result, the support of graphical interactions is partially language-independent (handling the user interactions affecting the predefined graphical constructs) and partially language-dependent (handling the specialized graphical constructs and the mapping to the abstract syntax).

The support provided by GMF-Tooling cover seven default graphical construct categories. These categories are detailed in TABLE III. However, GMF-Tooling also provides means to extend the categories with custom ones in order to handle language-specific constructs.

<table>
<thead>
<tr>
<th>GMFGen metamodel element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenDiagram</td>
<td>the root node containing all the elements of a diagram instance</td>
</tr>
<tr>
<td>GenMultifacetedNode</td>
<td>a generic node that can be directly contained by the diagram or by another node</td>
</tr>
<tr>
<td>GenAffixedNode</td>
<td>a node that will be affixed to another and thus considered as included in it</td>
</tr>
<tr>
<td>GenLabelNode</td>
<td>a label referencing an attribute not directly affiliated to the element in which it is graphically included (e.g. nested classifier label)</td>
</tr>
<tr>
<td>GenCompartment</td>
<td>a compartment included in a node</td>
</tr>
<tr>
<td>GenLink</td>
<td>a generic edge with a set of possible source nodes and target nodes</td>
</tr>
<tr>
<td>GenLabel</td>
<td>a label referencing an attribute of the element in which it is graphically included (e.g. name label)</td>
</tr>
</tbody>
</table>

The use case considered here only includes three UML elements: UML::Class, UML::Package, and UML::Association. Six of the graphical construct categories out of the seven provided by default in GMF-Tooling were sufficient to cover our use case: only the GenAffixedNode category was not used. Also, our use case only involves very common graphical interactions (drag & drop, edge redirection and simple label edition). Hence the default graphical construct categories provided by GMF-Tooling fully cover our
requirements and no additional development was needed for building custom categories.

However, the Diagram Interchange (DI) metamodel only distinguishes between DI::Shape, DI::Diagram, and DI::Edge elements, which correspond to GenMultifacetedNode, GenDiagram, and GenLink elements in the GMFGen metamodel, respectively. To allow mapping to more specific GMF-Tooling constructs, a UML profile was implemented to enrich the classification in the specialized DI metamodel. This simple UML profile, shown in Fig. 12, covers the four graphical constructs implemented in the GMFGen metamodel that are missing in the DI metamodel with the following stereotypes: an affixed shape stereotype for the GenAffixedNode element, a compartment stereotype for the GenCompartment element, a label stereotype for the GenLabelNode element, and an internal label stereotype for the GenLabel element.

![Fig. 12. A UML profile for the Diagram Interchange metamodel targeting an implementation in GMF Tooling.](image)

C. The Result of the Generative Processs

Fig. 13 gives an overview of the diagram editor resulting from this generative process. The generated Papyrus editor proves to be conforming to the DD-based specification in the following ways: a) the provided custom palette only implements the specified elements, b) the implemented shapes and edges are rendered consistent with specifications, and c) the diagram interactions – shape composition, edge redirection, and simple label editing – are constrained with respect to the specification and produce expected changes to the abstract syntax model.

Even though the result is promising, we will analyze in the next section the gaps between the diagram editor that is generated by this approach and an editor that is built using the traditional GMF process (e.g., the released Papyrus editor).

![Fig. 13. A simplified UML class diagram editor generated from a DI specialization model.](image)

VI. DISCUSSIONS

The approach presented in this paper exploits and assesses the potential of the recently released Diagram Definition standard as a framework for building a graphical editor for a given modeling language. An implementation of the approach is proposed, based on a case study restricted to a simple subset of the UML class diagram specification, and relying mainly on the GMF-Tooling framework. Building such an implementation for the solution allowed us to assess the potential and the gaps of Diagram Definition according to the different aspects of the architecture proposed in II.

A. Formally Specifying the Visual Grammar

The DI metamodel fulfills most of the requirements for formally specifying a simple subset of the UML visual grammar. The MOF refinement mechanisms (subsetting and redefinition) used in the UMLDI metamodel allow fine-grained specialization of the DI constructs (classes and associations). However, the current state of the UMLDI metamodel does not provide a detailed enough refinement of the visual grammar. We illustrated this issue earlier in Fig. 8 with the example of a class shape comprising several compartments, some of which being invalid according to the UML specification. This issue can be tackled by performing further refinement in the UMLDI metamodel. Eventually, the MOF refinement mechanisms proved to be sufficiently effective in fully constraining the visual grammar.

Indeed, the implementation used both the subsetting and redefinition mechanisms to refine the UMLDI metamodel (cf. III.B). The subsetting mechanism was used when one association end (the superset) is partially specialized by another (the subset). For example, when a shape composes both compartments and labels, the shape class defines a composition to each of the two concepts that subset the ownedElement-owningElement composition in DI. In contrast, the redefinition mechanism was used when one association (the redefining) is the exclusive specialization of another (the redefined). For example, the modelElement association end is typically redefined down the hierarchy and its type and/or multiplicity is often further restricted.

However, we found that building a fully constrained visual grammar specification requires a significant amount of time and lead to specialized DI metamodels with substantial sizes. Applying the same approach on a greater language specification, such as the complete UML specification, will undoubtedly imply a lot more complexity, volume, and development time than it is the case with the current informal approach. But this required additional effort is legitimate for fully formalizing the specification of a language and hence allowing automatically generating compliant tool support.

B. Mapping to the Abstract Syntax

The case study of our approach is restricted to have only one-to-one mappings between the concrete syntax and abstract syntax elements, thus allowing having a relatively straightforward exploitation and implementation of the DD-based specification. Yet, we show that this restriction does not prevent ambiguities from arising when trying to do the reverse
mapping from the graphical relationships in the specialized DI to the abstract syntax’s relationships.

This issue can be tackled by two different means. The first approach would be for the language designer to map every graphical relation in the DI metamodel to the appropriate relationship in the abstract syntax metamodel. Thus, the interpretation of the DD-based specification would be fully reverse-mapped, hence deterministic and unambiguous. The second approach would be to rely on the end user interactions for disambiguation. For instance, whenever the user intends to relate a graphical element on another, the diagram editor would compute on the fly the possible affected relationships in the abstract syntax metamodel that may correspond to the graphical relationship. If more than one relationship is found, the diagram editor would prompt the user with a list of the valid options and ask to select which one should be used.

Furthermore, practical cases are seldom restricted to one-to-one mappings between graphical constructs and abstract syntax elements. A good example is the UML::Association: the creation of a UML::Association must come along with the creation of two UML::Property elements, which are not necessarily contained by the UML::Association. This kind of behavior cannot be formalized — and hence automatically exploited — without a more elaborated reverse mapping. Therefore the future evolution of Diagram Definition should address this aspect and provides an additional mechanism for specifying reverse mapping.

C. Identifying the Graphical Element’s Behaviors

One of the most important features of DD is to map the concrete syntax to its corresponding graphical constructs. There are theoretically no boundaries to the diversity of graphical constructs that can be found in the world of graphical modeling languages. Nevertheless, it is reasonable to say that the graphical constructs of node and edge, together with the concept of diagram itself, can still be considered as a very common basis of such languages. The fact that these three graphical concepts are specifically identified within the Diagram Definition standard can hence be considered as legitimate.

However, as it is shown in the implementation of the case study proposed here, the single classification of these three graphical constructs is far enough for implementing tool support for a diagram such as a simplified UML class diagram. In this paper, the solution proposed to answer this issue is to implement a UML profile providing identification for the graphical constructs specific to the GMF-Tooling framework. This approach, being platform specific, is of course not acceptable to be followed in a standard. Therefore, we claim that the DI metamodel should not provide a richer classification of graphical constructs, but rather provide libraries of specific graphical constructs (through mechanisms like UML profiles) that can be reused by language-specific DI metamodels.

D. Specifying the Graphical Vocabulary

It is intended for the DD-based specifications to specify the rendering of a DI model by means of a model transformation that generates a corresponding DG model. Implementing this mechanism, even for a low-scale tool, requires some significant effort. Therefore, this is not part of our case study, meaning that while the graphical behavior of the diagram is specified using DD, the graphical rendering is not. Instead, we implemented the graphical rendering directly using the existing solution provided by GMF-Tooling. Nevertheless, we did discuss the possibility of such support for a DG-based diagram rendering within the GMF-tooling framework in section V.B.

Furthermore, the UML DD specification provided along with the DD standard, as an informative annex, includes a QVT transformation that transforms a UMLDI model into a DG model. This transformation is coupled with an SVG renderer that supports rendering DG models and hence allows graphically rendering a UML class diagram specified in UMLDI. Although such a renderer does not support user interactions – the main purpose of a model editor –, it tends to show that the mechanism suggested by DD is consistent and can help implement a model editor in the future.

VII. CONCLUSION

Standard diagrammatic languages are widely used in the industry, while developing and maintaining efficient tool support for such languages remain costly in time and resources. It is for now commonly accepted to use metamodeling techniques (e.g., the EMF framework) to specify and implement the abstract syntax of a graphical modeling language. However, a graphical modeling language consists also of a notation (also called concrete syntax) that is associated to the abstract syntax. In this area, the lack of a standard framework that allows language designers to build formal concrete syntax specifications raises several issues. The most important issue is that toolsmiths do not have the means to develop diagram editors that strictly comply with the language specification and that support diagram interchange.

The Diagram Definition standardized by the OMG provides a framework for specifying the concrete syntax of a graphical modeling language. In this paper we assess the potential of this standard in a practical context by using the framework for building a simple language specification and then trying to exploit it through an automatic generative process. We show that the use of the standard, as it is proposed in the UML 2.5 release, does not sufficiently meet the requirements for systematic implementation of compliant tool support. Indeed, the visual grammar definition lacks details, some graphical behaviors are missing, and the interpretation of the mapping with the abstract syntax raises some ambiguities.

Nevertheless, the potential of Diagram Definition is promising as we were able to generate a UML class diagram editor from the DD-based specification UMLDI with no major alterations (except manually specifying the rendering to graphics). However, the visual grammar specification had to be greatly enriched, and a UML profile was implemented to allow conforming to GMF-Tooling’s platform specific requirements.

Finally, the paper identified some directions for future work such as: capturing properly the mapping of the concrete syntax to the abstract syntax and defining what should be standardized about graphical construct behaviors and how to achieve that.
REFERENCES


