A Component-Based Approach for Specifying DSML's Concrete Syntax

Amine El Kouhen, Cedric Dumoulin, Sébastien Gerard, Pierre Boulet

A Component-Based Approach for Specifying DSML's Concrete Syntax

Amine El Kouhen
CEA LIST, Laboratory of Model Driven Engineering for Embedded Systems
Gif-sur-Yvette, France
amine.elkouhen@cea.fr

Cédric Dumoulin
University of Lille, LIFL CNRS UMR 8022
Cite scientifique - Bâtiment M3
Villeneuve d'Ascq, France
cedric.dumoulin@lifl.fr

Sébastien Gérard
CEA LIST, Laboratory of Model Driven Engineering for Embedded Systems
Gif-sur-Yvette, France
sebastien.gerard@cea.fr

Pierre Boulet
University of Lille, LIFL CNRS UMR 8022
Cite scientifique - Bâtiment M3
Villeneuve d'Ascq, France
pierre.boulet@lifl.fr

ABSTRACT

Model-Driven Engineering (MDE) encourages the use of graphical modeling tools, which facilitate the development process from modeling to coding. Such tools can be designed using the MDE approach into meta-modeling environments called *metaCASE tools*.

It turned out that current metaCASE tools still require, in most cases, manual programming to build full tool support for the modeling language, especially for users' native methodologies and representational elements and propose limited possibilities in terms of reusability. In this context, we propose *MID*, a set of meta-models supporting the easy specification of modeling editors by means of reusable components and explain how representational meta-modeling is carried out with it.

Categories and Subject Descriptors

I.6.5 [**Computing Methodologies**]: Model Development

; D.2.2 [**Software Engineering**]: Design Tools and Techniques

; D.2.13 [**Software Engineering**]: Reusable Software

Keywords

MetaCASE tools, Component-based metamodeling, Visual languages, Model-Driven Development (MDD), Reusability, Concrete Syntax

1. INTRODUCTION

As part of its Model-Driven Architecture (MDA) initiative, the Object Management Group (OMG, http://www.omg.org) - an international consortium representing numerous industrial and academic institutions - has provided a comprehensive series of standardized technology recommendations in support of model-based development of both software and systems in general. These cover core facilities such as meta-modeling, model transformations, and general-purpose and domain-specific modeling languages. A key component in the latter category is UML (the Unified Modeling Language), which has emerged as the most widely used modeling language in both industry and academia. A number of tools supporting UML are available from a variety of sources. These are generally proprietary solutions whose capabilities and market availability are controlled by their respective vendors. Consequently, some industrial enterprises are seeking open-source solutions for their UML tools. To respond to this requirement, a new graphical editor called *Papyrus*, was accepted by the Eclipse Project MDT in August 2008. This graphical editing tool for UML2 is based on Eclipse and uses the Eclipse graphical modeling Framework (GMF).

Papyrus is a tool consisting of several editors, mainly graphical editors but also completed with other editors such as textual-based and tree-based editors. All these editors allow simultaneous viewing of multiple diagrams of a given UML model. However, when such diagrams were specified, we found common problems at different levels. The first common point relates to the redundant elements in all UML diagrams, such as *Comments* or *Constraints* elements that are presents in all Papyrus diagrams. The second common point relates to some specific diagrams as *Package Diagram*, which is composed of a *Class Diagram* subset (Package, import, merge ...). The other common point relates to the graphical variation of several features. For example, a *Class* in the class diagram does not have the same graphical representation as in the composite structure diagram. The same thing for the *Actor* element in the use cases diagram and

other diagrams. These elements have the same semantics in the UML model, sometimes they have the same graphical structure, but they are represented with different shapes. This statement raises a real issue of reuse when specifying diagrams.

To explain these issues, we evaluated the technologies currently used to specify Papyrus diagrams [5]. It turns out that the main reason for these gaps is the lack of reusability in this kind of technology. This results in manual copies in all diagrams, thus increasing the risk of error, problems of consistency, redundancy in the specification and the difficulty of maintenance.

At a high level of abstraction, the study of these tools, and especially GMF, allows us to identify some needs and criteria in terms of reusability, graphical completeness, model consistency and maintainability of diagrams specifications. Compliance with these criteria led us ultimately to produce an alternative meta-tool based on a set of meta-models called *MID* (**M**etamodels for user **I**nterfaces and **D**iagrams), to rapidly design, prototype and evolve graphical editors for a wide range of visual languages. We base MID's design on three overarching requirements: graphical completeness, ease of use and simplicity of (de)composition of diagrams editors for a better reusability. For that, we take advantage from MDE benefits, Component-based modeling and an inheritance mechanism to increase the reuse of editors' components. The main goal of this work is the specification and the generation of UML diagram editors of Papyrus [7], from reusable pre-configured components.

For all of this, we consider it being useful to present the basics of visual languages as well as the study we made on tools and modeling methods in section 2. This section ends with a summary of issues arising from this study and a proposal for criteria to be met by our solution. Then, section 3 describes, our approach to design graphical editors based on our proposal: MID meta-models. In section 4, we validate our approach through examples using our proposed inheritance mechanism to reuse graphical components of a diagram.

2. FOUNDATIONS AND RELATED WORKS

In this section, we present the state of the art of graphical modeling environment. We begin by presenting visual representation foundations to extract concepts that describe a diagram and then studying several existing methods and tools for specification and generation of graphical editors for diagrams. This study has identified several issues, which we expound as evaluation criteria for our approach.

2.1 Visual Languages Basics

Visual representation is one of the oldest forms of knowledge representation and predates conventional written language by almost 25,000 years [18]. Visual representations play significant roles in scientific reasoning [17].

According to [15], elementary components of a visual representation are called visual notations (visual language, diagramming notations or graphical notations) and consist of a set of graphical symbols (**visual vocabulary**), a set of compositional / structural rules (**visual grammar**) and definitions of the meaning of each symbol (**semantics**). The visual vocabulary and visual grammar form together the **concrete** (or **visual**) **syntax**.

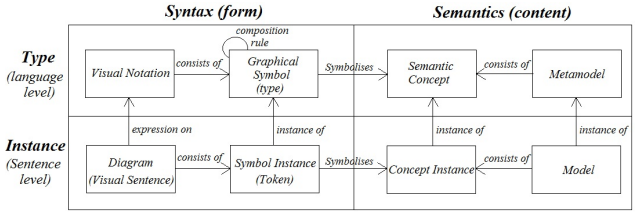

Figure 1: The nature of a visual notation [15]

In the literature, numerous definitions can be found for the concept of diagram. The widely accepted ones, are that of Kosslyn [10], Larkin [11] and Tversky [19]: *Diagrams are an effective medium of human thinking and problem solving. Diagrams are thus bi-dimensional, geometric, symbolic and human-oriented representations of information; they are created by humans for humans. They have little or no value for communicating with computers, whose visual processing capabilities are primitive at best* [16].

Graphical symbols are used to symbolize (perceptually represent) semantic constructs, typically defined by a metamodel [8]. The meanings of graphical symbols are defined by mapping them to the constructs they represent. A valid expression in a visual notation is called a **visual sentence** or **diagram**. Diagrams are composed of symbol instances (tokens), arranged according to the rules of the visual grammar [15]. Such distinction between the content (semantics) and the form (syntax: vocabulary and grammar), allows us to separate the different concerns of our proposition. These definitions are illustrated in fig. 1.

The seminal work in the graphical communication field is Jacques Bertin's *Semiology of Graphics* [1]. Bertin identified eight elementary visual variables, which can be used to graphically encode information (Fig. 2). These are categorized into planar variables (the two spatial dimensions x,y) and retinal variables (features of the retinal image).

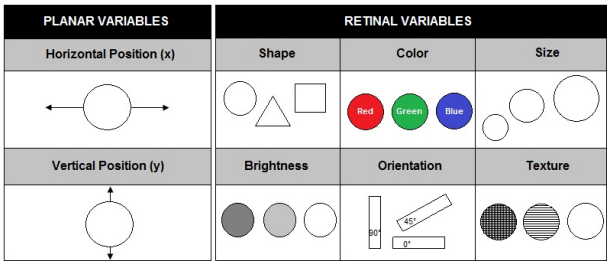

Figure 2: Visual variables [1]

The set of visual variables define a *vocabulary* for graphical communication: a set of atomic building blocks that can be used to construct any graphical representation. Different visual variables are suitable for encoding different types of information. The choice of visual variables has a major impact on cognitive effectiveness as it affects both speed and accuracy of interpretation [3, 13, 21].

A Diagram is defined as a planar graph, which has hybrid languages capabilities [2]. In the graph theory field, a pair G=(V, E) with E⊆E(V) is called **graph** (on V). The elements of V are the **vertices** of G, and those of E the **edges** of G. In literature, vertices are also called nodes or points; edges are called lines or links. These two concepts and others are the main ones in our visual grammar definition. We present them in section 3.

2.2 Evaluation Criteria

Many frameworks, meta-tool environments and toolkits have been created to support the development of visual language environments. We conducted an evaluation of the technologies currently used to specify the diagrams in Papyrus and other techniques and tools to do so in the state of the art. It turns out that the main cause of such limitations is the lack of reusability mechanism in this kind of technology. This results in manual copies in all diagrams, thus increasing the risk of error, problems of consistency, redundancy in the specification and the difficulty of maintenance. To understand these issues, we offer an overview of our tools evaluation. The diagrams specification methods have been widely discussed in [5, 2]. We summarize them in the following categories:

1. Code-based specification. As GEF, UMLet and Graphiti. This kind of tools construct graphical representations from a code description.
2. Proprietary languages-based specification. This category of tools uses meta-description languages to specify graphical editors, as MetaEdit+, which uses the GOPPRR metamodeling language [9] and GME [12], which uses a meta-description to specify the abstract and concrete syntaxes for a specific domain.
3. Specification based on graph grammar. This kind of tools is based on the graph grammar definition and allows specifying diagrams from this grammar (e.g. Visual DiaGen [14], AToM^3 [4]).
4. Other tools allow editor specifications; they are exclusively graphical drawers i.e. they care only about the graphics without giving meaning to them: semantics mapping-less (e.g, Microsoft Visio or Dia).
5. There are other methods and tools, which have recently adopted MDE approaches; we distinguish two major categories in this specification method :
   - Tools based on UML profiles as Papyrus [7], IBM RSA and MagicDraw, which propose to create visual representations for profiles' metaclasses (UML Stereotypes mechanism).
   - Tools based on DSML's. Domain-Specific Modeling Languages are more specific to domain requirements, giving users specific concepts to their occupation and their expertise. GMF tooling, Spray, TopCased Meta and Obeo Designer are examples of tools supporting this kind of specification. They describe (via models) the concrete syntax and associate it to a specific domain metamodel. This kind of technology, including GMF is used in Papyrus for several reasons (ease to manage compared to a model code, open-source technology, ease of integration with the Eclipse ecosystem ...). However, these tools still require programmatic interventions for adaptation. Many other shortcomings are detected, mainly in terms of reuse and tools rigidity [5].

This article focuses on the criterion of reusability that we consider very useful in the context of diagrams specification. Among the forms of reuse, we can cite the separation of concerns, inheritance and overriding.

Most of diagrams specification methods mix concerns. The most common form of this mixture is that of form and content (visual representations and semantics). For example, in the case of diagram specification using GME or MetaEdit+, these tools allow creating the specific concepts and their associated representations in the same repository. This weakens the required loosely coupling relationship between the semantics and the graphical aspects, which limitate the reuse of the concrete syntax. The same problem is observed with Obeo Designer, which allows graphical/semantics association in the same model.

Other form of mixing is between the visual vocabulary and visual grammar definition. Most of the tools offering the separation of the graphical part from the semantic one, as GMF tooling, TopCased-Meta and even standards like Diagram Definition (OMG, http://www.omg.org/spec/DD/), fail to separate the two graphical syntax concerns, which are visual vocabulary (shapes, colors, styles ...) and visual grammar (structure and composition of representations).

We found another problem with GMF or Obeo Designer: few opportunities (or lack) of elements specification reuse, which usually results *redundant specifications* at the **model-level** and *redundant classes* at the **code-level**.

The design of Papyrus brings up an important need in terms of diagrams definition reusability. It would be nice to describe a library of concrete syntax, independent from the semantics, to be reused (if necessary) in different diagrams, to factorize common concrete syntax in common definitions and variations in specific definitions based also on the common definitions. This will allow us to define diagrams independently from each other with the possibility of reusing and redefining (overriding) common elements.

3. METAMODELS FOR USER INTERFACES AND DIAGRAMS (MID)

The aim of our work is to design diagram editors and to allow reusing parts of such design. For that, we propose to use a Model-driven approach, to ensure the independence to technology, ease the maintenance and enable better sustainability.

To improve reusability, we propose a component-based approach. This approach aims to take advantage of encapsulation (ease of maintenance and composition) and the benefits of interfacing (interfaces naming mechanism). In addition, our approach allows the reuse through inheritance: a component can inherit from another one and it can also override

some of its properties (style, structure, behaviour...).

The first advantage of this approach is the independence from the semantics, which allows assembling graphical elements differently from a diagram to another. Another advantage is the ease of the diagram specification maintainability. By avoiding redundancy and duplication at the generated code, we can facilitate the maintainability of the diagram specification, but also by providing a mechanism that allows us to reflect any change in a definition to all places using it. The component approach is promising in this context. By defining the composition (structure) of a graphical element and the encapsulation of its interfaces, it is easy to maintain and modify it.

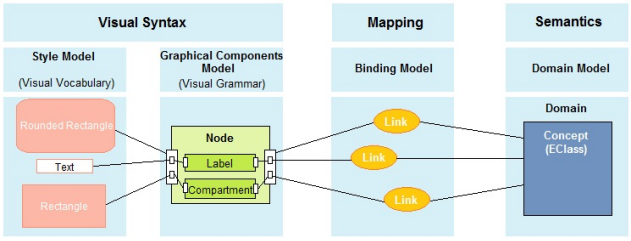

Figure 3: MID: Involved Artifacts relationship

Figure 3 shows the linkage of the meta-models involved in our proposal. First, we separate the domain content (**semantics**) and the form (**visual syntax** or concrete syntax) of a diagram at a high level of abstraction (language level). The Semantics is out of scope of our paper, it is widely treated in tools and technologies like EMF/Ecore. The form is separated into two parts : the **visual vocabulary** (different variables of shape, color, size ...) and the **visual grammar** that describes composition rules of visual representations. The link between the syntax and the semantics is also specified in a separate "binding" model. Thus, our proposal is made of several meta-models, each one used to describe one concern: a visual grammar meta-model, a visual vocabulary meta-model and a mapping meta-model.

3.1 Component Metamodel

Our goal is to specify and compose the elements of the diagram editor with an approach based on the characteristics of the components. We follow a model-based approach for modeling diagram components to solve two problems : components heterogeneity and their composition techniques.

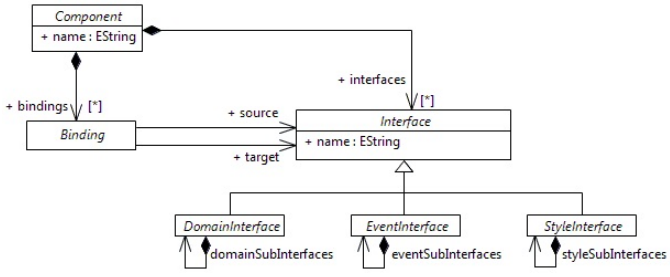

Figure 4: Component Metamodel

The component concept is the main concept of our set of meta-models. A component could have interfaces (in our context there are three types of interfaces: *domain*, *style* and *event interfaces*) and the bindings between such interfaces. Interfaces are used as an attachment point between (sub)components and the other concerns (semantics, behaviours and styles).

3.2 Visual Grammar: Diagram Composition

The visual grammar is used to describe the structure of diagrams' elements. This description is hierarchical: a root element can contain other elements. We propose two main types of elements: *Vertices* to represent complex elements of diagrams and *edges* to represent links between complex elements.

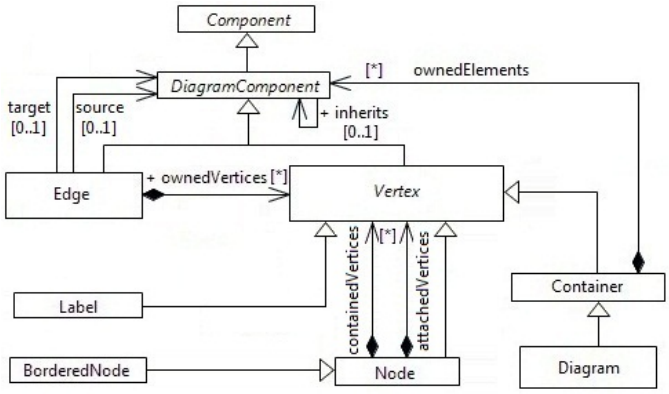

Figure 5: Diagram Elements

Vertex is node abstraction and shape formalization [6], it consists of main nodes (top nodes), sub-nodes (contained vertices in figure 5) and attached nodes (nodes that can be affixed to other nodes). A label is a vertex that allows access to nodes textual elements via their accessors (getters and setters). This will synchronize the data model with text value represented. A Bordered node is a node that can be affixed to other nodes. Containers (Compartments) are specific nodes that contain diagram elements. A Diagram is itself a container. An Edge is a connection between two diagram elements, this relationship could be specified semantically (in the domain metamodel) or graphically and could be more complex than a simple line (e.g, a data bus between two devices).

The meta-model in figure 5 represents diagrams main concepts. It allows designing all hybrid visual languages [2], which we use in the MDE field like UML, Petri Net, BPMN...

Edges and nodes have both the ability to inherit from each others. When a diagram component inherits from another, it recuperates all its properties (structure, style and behaviour). If the inheriting component has an element with the same name as the inherited component, this is interpreted by an overriding and then we can override the structure, style and behaviour. This feature maximizes components reuse and allows creating other derivatives components. The proposed rules of graphical inheritance are resumed in the algorithm 1.

Algorithm 1 MID: Graphical Inheritance mechanism

```
if Component B inherits from Component A then
  B gets all sub-components and properties of A;
  if an element of B has the same name as an element of
  A then
    The element of B override the element of A;
    The interfaces can also be overrided;
    if we need to update this element then
      we should redefine its interface;
    else if we need to delete this element then
      we do not specify its interface;
    else if we need to add an element then
      we should add its interface;
    end if
  end if
end if
```

Visual grammar elements only represent the structure and should be associated to a visual vocabulary describing its rendering.

3.3 Visual Vocabulary: Visual Variables

Visual vocabulary allows describing the graphical symbols (visual representation) of diagrams' elements. This description is composed of different visual variables; we regroup all of them in the *Style* concept (fig. 6) representing the shape, color, size, layout...

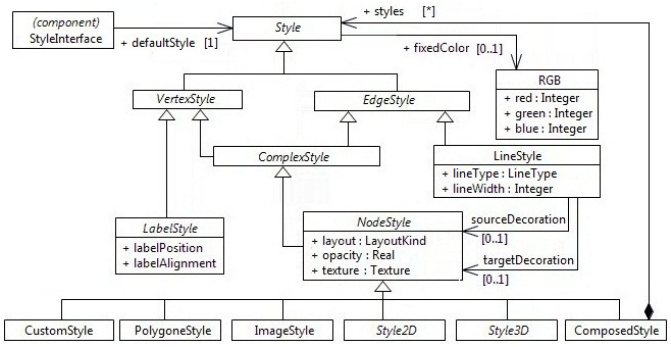

Figure 6: Visual variables(Styles) description

All diagram components are associated via their *style interfaces* to visual vocabularies represented in the meta-model by the concept of **Style**. As other characteristics of diagrams elements, this relationship can be reused and overriden through the proposed mechanism of inheritance.

The *Vertex Styles* are characterized by the layout attribute, which represent the different arrangement rules in the host figure. Vertex styles are decomposed into two main categories: node styles (**NodeStyle**) and label styles (**LabelStyle**). Node styles represent shapes (2D and 3D figures and iconic representations like images). We propose ten default shapes in our meta-model, and we let users to create their own shapes with polygons, images and custom styles (code implementation). Label styles specify label alignment, position and label type. The *Edge Styles* could be simple lines (**LineStyle**) or more complex shape (**ComplexStyle**).

3.4 Domain Mapping

The mapping meta-model (fig. 7) allows binding the different graphical elements (nodes and edges) to their corresponding semantic concepts. A graphical element can be mapped to one or more semantic elements. This is done through the concepts **NodeBindingSet** and **EdgeBindingSet**, each containing a set of associations **NodeBinding** or **EdgeBinding**.

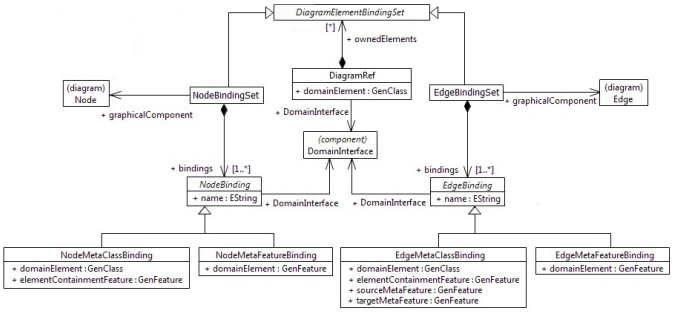

Figure 7: Semantics Mapping

Currently, the description of the mappings serves as entry point to the full description of the diagram. This is represented by the element **DiagramRef**, which contains all associations. This approach allows us to reuse graphical representations in different diagrams with different semantics. For example, in a UML class diagram, it is possible to use the same representation for both concepts *Class* and *Interface*. We use this to increase the reusability of graphical components.

3.5 Representation formalism

For simplicity, we propose a graphical formalism to present our concepts. This formalism allows to see graphically the diagrams specification instead of textual or tree-based form. Thus, we propose a concrete syntax for our metamodels. Note that our tools can also define this formalism with the same concepts (auto-description).

The figure 8 shows an example of a component specification with the graphical view (top right) and its equivalent in tree-based view (top left). Both views allow the result in the bottom of the figure.

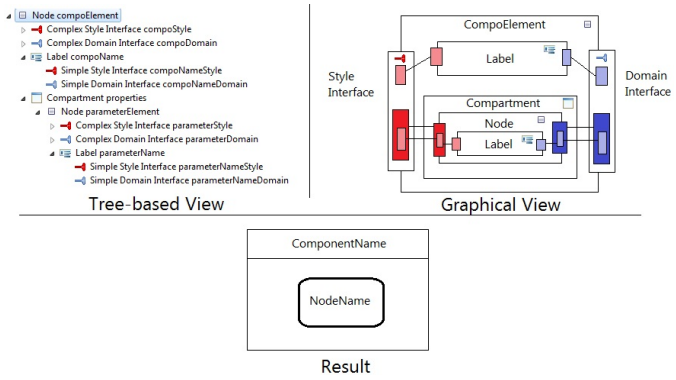

Figure 8: MID Graphical Formalism

4. VALIDATION

To validate our proposal, we have developed a chain of transformations allowing the full generation of designed editors code. Note that MID meta-models are completely independents from technological targets. In the actual implementation, we choose GMF as technological target.

We illustrate the advantages of our approach on an example specifying the UML *Classifier* element to show the reusability through inheritance. Then, we validate our approach through a case study.

4.1 Reuse by Inheritance

We chose as an example the UML concept *Classifier*. This abstract concept is the basic element of several concepts (*Class*, *Interface*, *Component*...). We want to specify the graphical appearance of the item *Classifier* and use inheritance and overriding to specialize this specification.

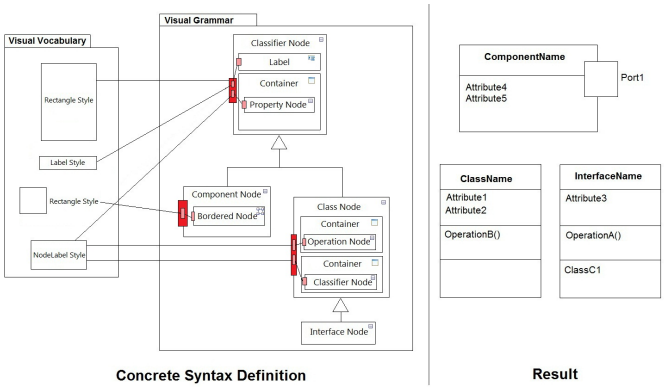

Figure 9: Reuse by Inheritance

The basic element of *Classifier* consists graphically of a label followed by a compartment that contains properties. To specify a *Component*, we have simply to inherit from *Classifier* and add to its structure, a border node representing the component ports (attached on borders).

To Specify the graphical elements *Class* and *Interface*, we have simply to inherit from *Classifier* and add to its structure two other compartments, the first for operations and the other one for nested classifier. In this example, and to simplify, the Interface inherits from the graphical definition of the Class to show the graphical similarity between the two concepts. The figure 10 shows the generated result.

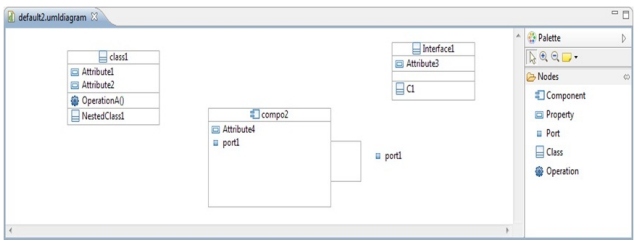

Figure 10: Generated editor for this example

The example below shows the overriding of inherited elements. The component "**Node B**" inherits from the component "**Node A**". Both have sub-components named "Node x", in this case the element *"Node x" of B* overrides the description of *"Node x" of A* (initially represented by an ellipse).

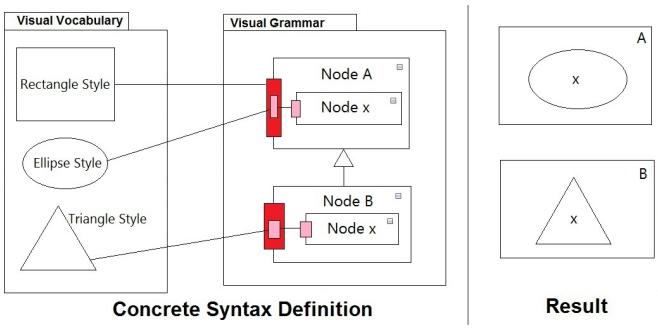

Figure 11: Example of graphical overriding

4.2 Chain of Transformation MID→GMF

To validate quickly our approach we have developed a transformation chain to generate Java code for diagram editors using GMF. The transformation chain allows us to move from a modeling workspace to another. Each workspace is at a level of abstraction and detail higher than the other. Intermediate models are described by meta-models to add technical details and technology needed to code generation. We took care to introduce technological details the latest possible. These details appear in the latest model of the chain (GMFGen Model).

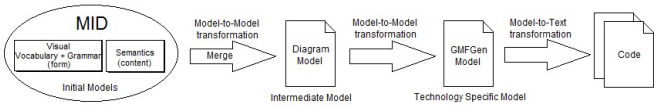

Figure 12: Chain of Transformation MID→GMF

This transformation chain allows us to generate the entire graphical editor. The tool user has only to run the application. The generated editor allows manipulating domain-specific concepts with graphical representations specified at the model level.

Currently, our approach allows reusability at the level of diagrams design. However, the chosen target technology (GMF) does not allow the reuse at the level of the generated code. It remains duplicated in the code-level, but with a single definition at our meta-models.

4.3 Case Study

We choose for the case study, to design the BPMN diagram and the UML state-machine diagram. Because of their graphical resemblance, this choice aims to show the level of components reusability in both editors.

4.3.1 BPMN Workflow Diagram

The Business Process Management Initiative (BPMI) has developed a standard Business Process Modeling Notation (BPMN). BPMN is dedicated to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes,

to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes [20].

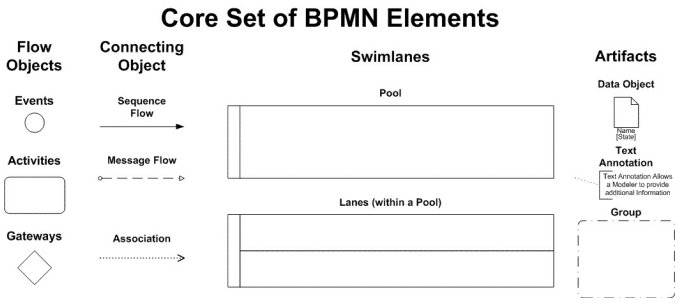

Figure 13: Graphical elements of BPMN

BPMN defines a Business Process Diagram (BPD), which is based on a flowcharting technique tailored for creating graphical models of business process operations. A Business Process Model, then, is a network of graphical objects, which are activities (i.e., tasks) and the flow controls that define their order of performance [20]. In terms of concrete syntax elements, there are four basic categories of elements: Flow Objects, Connecting Objects, Swimlanes, and Artifacts. The symbols corresponding to them are summarized in Figure 13.

For this example, we have created a simple metamodel that includes a subset of the language concepts. This metamodel is not meant to be a realistic representation of BPMN (this is out of the scope of this study). A complete specification of the BPM notations and semantics can be found in (OMG, http://www.omg.org/spec/BPMN/2.0/).

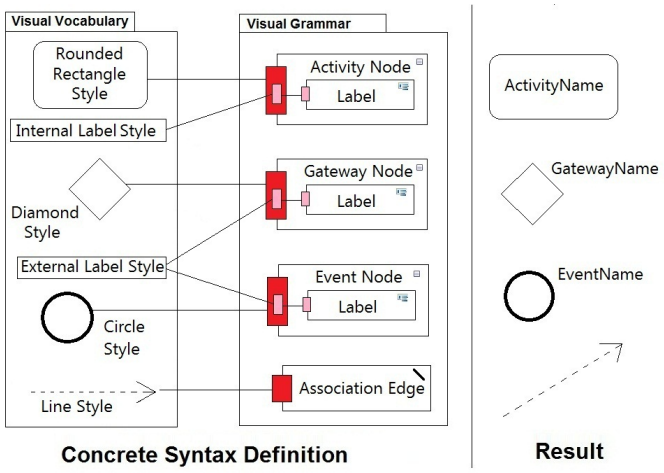

Figure 14: Specification of BPMN Diagram

We define thus BPMN diagram elements by mapping each graphical component to a style via its style interface for defining different visual variables. We find graphical similarities between events (start, intermediary, end) and gateways (OR, AND, XOR). For this reason, we use the inheritance mechanism between these elements by overriding their styles for specific needs.

4.3.2 UML State-Machine diagram

State machines can be used to specify behaviour of various model elements. For example, they can be used to model the behaviour of individual entities (e.g., class instances). The state machine formalism described in this sub clause is an object-based variant of Harel statecharts (OMG, http://www .omg.org/spec/UML/).

UML state machine diagrams depict the various states that an object may be in and the transitions between those states. In fact, in other modeling languages, it is common for this type of a diagram to be called a state-transition diagram or even simply a state diagram. A state represents a stage in the behaviour pattern of an object, and like UML activity diagrams it is possible to have initial states and final states. An initial state, also called a creation state, is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

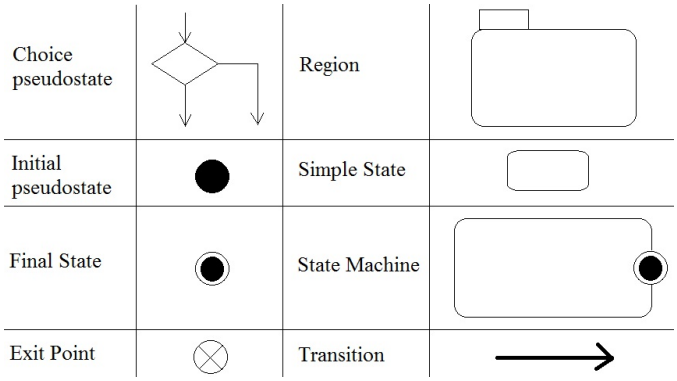

Figure 15: Graphical elements of UML SM

For this example, we have used the UML2 meta-model implementation within Eclipse (also known as the "UML2 Component"). This component has become the *de facto* standard implementation of the UML2 meta-model (note that it is also the basis for the UML2 tool suites provided by IBM).

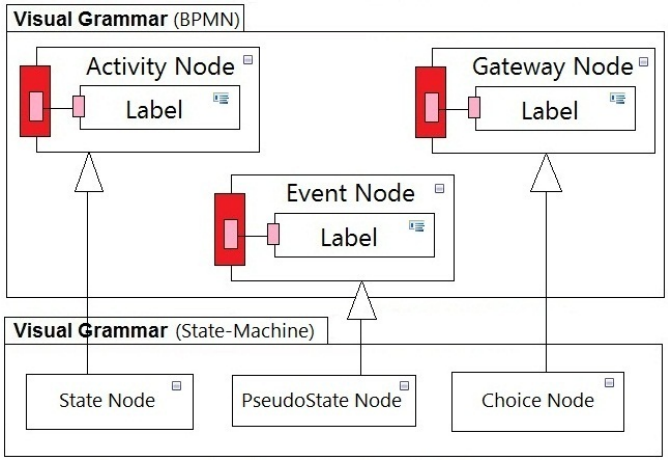

Figure 16: Specification of State-Machine Diagram

As BPMN Workflow Diagram, we define the UML State Machine Diagram with the same process. Because of the graphical similarity between BPMN and UML State Machine, we reuse most of components already defined in the first example (more than 80% of BPMN components). We defined the specificities of the State Machine diagram, using the inheritance mechanism and the overriding of components. This allowed us to gain a considerable time of development in the second example.

The following screenshot show respectively the BPMN and UML State Machine graphical editors generated from the specifications presented above.

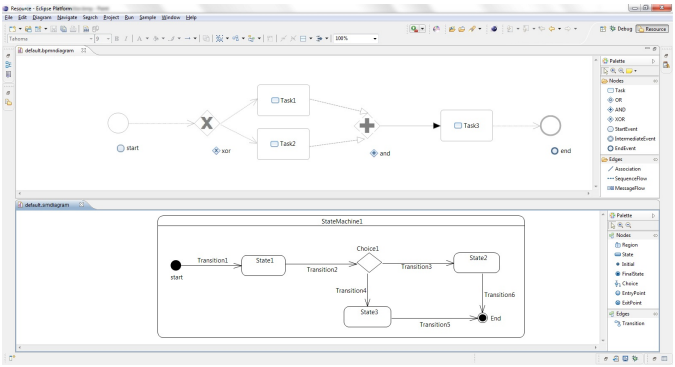

Figure 17: Generated editors: BPMN editor (top side) and State Machine editor (bottom side).

The full design and generation of BPMN and UML state machine editors, allow us to perform a first validation of our approach. We complete this validation by an evaluation according to the criteria identified in Section 2.2 of this Paper.

The specification method chosen for our approach is based on models. This approach allowed us to benefit the undeniable advantages of MDE into the editors' development cycle. These benefits are reflected in multiple aspects, such as ease of specification and technology independence, which allow a greater collaboration and flexibility in the metamodeling cycle of editors.

The reusability was since the beginning of our research work, the most important criterion and the most sought. This criterion motivated us to seek methods that allow more reuse of specification models. For this reason, we choose to introduce the concept of component-based metamodeling to specify graphical editors. Component-based approach ensures a better readability and better maintenance of models. It is particularly useful for teamwork and allows industrializing software development. Reusability of a component saves significant productivity gain because it reduces development time, especially as the component is reused more often.

Through the examples presented in Section 4.3, we validated our approach in terms of reusability: This approach allowed us (in State-Machine example) to **reuse more than 80%** of components created in BPMN example, which is not negligible. This approach allowed us to define more easily, editors' specificities with a model-driven approach and without any need to redefine or manually program changes, which increases the level of maintainability of editors generated with our solution.

The graphical completeness is defined by the capability to use fully the shape variable (the use of any kind of shapes : complex, composites, 2D/3D...). Unlike tools we have evaluated in [5], our metamodels have a great capacity to use the full range of these variables; we were inspired by tools based on graph grammar and their approaches to define those variables. We have proposed also representation methods used in the other tools like SVG representations, and other predefined figures : we propose around ten default shapes in our graphical metamodel, and we let the possibility to users to create their own shapes with polygons, images and the custom styles (classes). We solve some problems identified in existing tools and methods on the industry as in the literature. For example, the specification of complex diagrams editors as sequence diagram at a high level of abstraction without the need for manual programmatic intervention.

Unlike existing tools for diagrams specification, we separate the editor's aspects and concerns initially between the semantic and the graphical aspect, then we separate the two graphical aspects, which are the visual vocabulary (visual variables) and the visual grammar, which represents the composition rules of diagrams. Subsequently, it's important to create another part that would make the mapping between the different aspects, in particular between the semantic and the graphic. The separation of concerns is also carried out in the transformation chain, by introducing several intermediate level models and by delaying the introduction of technical details in the latest models of the chain, which allows a better maintainability of the transformation chain in case of a change in metamodels. A strong separation of concerns allows a better reuse and maintenance of models, it decreases development costs in terms of maintenance time in case of changes in these models and should allow designing new applications by assembling existing models. It also allows to reuse a complete diagram description with another domain model.

5. CONCLUSION

In this article, we present an approach based on MDE and components modeling, allowing the easy specification of diagram graphical editors at a high level of abstraction, in order to model, reuse, compose and generate code. In our proposal, we focus on the component concept, to describe and then assemble concepts emerging from visual languages. First, we present the definitions and theoretical foundations of visual representations and the component-based metamodeling approach while positioning our work in relation to different tools and technologies available in the industry and in the literature.

In our approach, we promote the use of reusable components (with composition, encapsulation, inheritance...) and a strong separation of concerns (domain, graphical element, styles). This increases the reusability of the editors and brings the benefits of the MDE paradigm as models verification/checking or the ability to choose target technologies through model transformation techniques.

In MID, we solve some problems identified in existing tools

and methods on the industry as in the literature. For example, the specification at a high level of abstraction without the need for manual programmatic intervention, the separation of concerns, the graphical effectiveness and finally editors reusability, which was among the major problematic of our research work. To validate our approach, we have developed a transformation chain targeting the GMF technology (GMFGen), which allows in turn generation of functional editor's code. This allows us to successfully design diagrams by reusing existing components, and to generate their implementation We validated our approach on several diagrams.

Our approach presents many advantages. First, through the reuse of model: the models are theoretically more easily to understand and to manipulate by business users; which corresponds to a goal of the MDE. Secondly, this reuse saves considerable gain of productivity through ease of maintenance of components; it allows better teamwork and helps for the industrialization of software development: it is possible to build libraries of components, and then build the diagram by assembling these components.

Briefly, we can say that our approach opens a new way that shows promises for wider use of modeling tools and automatic generation of applications. Compared to the current development technologies, the promises of this approach are large through the ability to create complex applications by assembling existing simple model/components fragments, and especially the possibility for non-computer specialists, experts in their business domain, to create their own applications from a high-level description using an adapted formalism, easy to understand and manipulate for them.

In the current state of our research, many studies are still required to reach a full generation of modeling tools. First, we need to finalize the description and generation of all graphical editors of Papyrus with our approach. Finally, we need to define other meta-models that allow description of the other parts of such tools (Tree editors, tables/matrices, properties views...) following the same approach of component reuse and inheritance.

6. REFERENCES

[1] J. Bertin. *Semiology of graphics : diagrams, networks, maps*. University of Wisconsin Press, Madison, Wisconsin, 1983.

[2] P. Bottoni and A. Grau. A suite of metamodels as a basis for a classification of visual languages. In *IEEE Symp. on Visual Languages and Human Centric Computing*, pages 83 –90, sept. 2004.

[3] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.

[4] J. De Lara and H. Vangheluwe. Using atom3 as a meta-case tool. In *4th International Conference on Enterprise Information Systems (ICEIS), April 2002, Ciudad Real, Spain*, pages 642–649, 2002.

[5] A. El-Kouhen, C. Dumoulin, S. Gérard, and P. Boulet. Evaluation of modeling tools adaptation. Technical report, CNRS, 2011. `http://hal.archives-ouvertes.fr/hal-00706701`.

[6] F. Fondement. Documentation succincte sur gmf. Technical report, Université de Haute Alsace, 2008. `http://fondement.free.fr/lgl/courses/mde/documentation_succinte_gmf.pdf`.

[7] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic. Papyrus: A uml2 tool for domain-specific language modeling. In *Model-Based Engineering of Embedded Real-Time Systems*, Lecture Notes in Computer Science. 2011.

[8] ISO/IEC. 24744: Metamodel for development methodologies, 2007.

[9] S. Kelly, K. Lyytinen, and M. Rossi. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 1–21. 1996.

[10] S. M. Kosslyn. *Image and Mind*. Harvard University Press, 1980.

[11] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65 – 100, 1987.

[12] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44 –51, nov 2001.

[13] G. L. Lohse. A cognitive model for understanding graphical perception. *Hum.-Comput. Interact.*, 8(4):353–388, Dec. 1993.

[14] M. Minas. Visual specification of visual editors with VisualDiaGen. In *Applications of Graph Transformation with Industrial Relevance, Proc. 2nd Intl. Workshop AGTIVE, Charlottesville, USA*, 2003.

[15] D. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 2009.

[16] D. Moody and J. Hillegersberg. Evaluating the visual syntax of uml. In *Software Language Engineering*, Lecture Notes in Computer Science. 2009.

[17] L. Perini. Visual representations and confirmation. *Philosophy of Science*, 72(5):913–926, 2005.

[18] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

[19] B. Tversky. Cognitive principles of graphic displays. In *AAAI 1997 Fall Symposium on Reasoning with Diagrammatic Representations*, November 1997.

[20] S. A. White. Introduction to bpmn. Technical report, 2009. `http://www.omg.org/bpmn/Documents/Introduction_to_BPMN.pdf`.

[21] W. Winn. Learning from maps and diagrams. *Educational Psychology Review*, 3:211–247, 1991.