Experimentation of a Graphical Concrete Syntax Generator for Domain Specific Modeling Languages

Blazo Nastov
ANSYS
31 PUBLICATIONS **78** CITATIONS

Francois Pfister
IMT Mines Alès
32 PUBLICATIONS **143** CITATIONS

Experimentation of a Graphical Concrete Syntax Generator for Domain Specific Modeling Languages

Blazo Nastov[1], François Pfister[1]

1. LGI2P, Ecole des Mines d'Alès, Parc Scientifique G. Besse, 30000 Nîmes, France {Blazo.Nastov, Francois.Pfister}@mines-ales.fr

ABSTRACT. *Graphical Domain Specific Modeling Languages (DSML) are alternatives to general purpose modeling languages e.g. UML or SysML. They describe models with concepts and relations specific to a domain. Defining such languages consists of defining an abstract syntax and a graphical concrete syntax accompanied by a correspondence mapping between the elements of each one. Such process is composed of two phases: the abstract syntax definition and the concrete syntax definition. This paper describes concepts and mechanisms allowing to guide and to assist an expert from any engineering domain to define and formalize the concrete syntax of a graphical DSML considered as relevant in this domain. We define multiple classifications of the abstract syntax elements based both on the abstract syntax and on the concrete syntax. Grounded on those classifications, we present how a part of the concrete syntax can be generated automatically from an abstract syntax by a graphical role election.*

KEYWORDS: *MDE, DSML, model, metamodel, languages engineering*

1. Introduction

Complex system engineering is an approach for designing complex systems based on creating, manipulating and analyzing various models. Each model is related to and is specific to a domain (e.g. quality model, requirements model or architecture model). Classically, models are the subject of study of Model Driven Engineering (MDE) (Kent 2002) and they are nowadays built by using, and conforming to Domain Specific Modeling Languages (DSMLs). Creating DSML consists in defining its abstract and concrete syntaxes. An abstract syntax is represented by a metamodel composed of classes representing modeling concepts and relationships between classes representing relations and dependencies between modeling concepts. The literature highlights two ways for defining an abstract syntax. Either extending UML (OMG 2011) (UML profile (Fuentes-Fernández & Vallecillo-Moreno 2004)) or deriving a metamodel directly from MOF (MOF 2002). Various *concrete syntaxes* define the representations of modeling concepts which are either graphical or textual.

This paper describes an experimentation of concepts and mechanisms allowing to guide and to assist an expert from any engineering domain to define and formalize a graphical concrete syntax for a given DSML considered as relevant in this domain.

A graphical concrete syntax is composed of graphical elements, each one representing how a given abstract syntax element (class or relationship) is graphically rendered for the end user. Basically, classes are represented as diagram nodes with one exception, representing a class as diagram edge detailed furthermore in the paper, and relationships are represented as diagram edges. Such representations can be refined by size, shape, role, color, etc. Once defined, they are mapped to abstract syntax elements. The mapping is called *correspondence mapping*. Together, the abstract syntax, the concrete syntax and the correspondence mapping, form a DSML syntax (Kleppe 2007). Such syntax allows creating models seen as graphical representations of a part of a modeled system.

The correspondence mapping is usually source of errors and abstract syntax elements may have inconsistent graphical representations, for instance visual information representing “node” mapped to relationship. This is either interpreted as a mapping error, or the considered relationship is excluded from the generated editor palette. We aim to reduce such errors based on possible graphical representation of modeling concepts which we refer to as “graphical roles”. We classify abstract syntax elements considering low levels of graphical representation, distinguishing if a given element would be graphically represented as node or edge or if it would not be graphically represented at all. High level graphical representation such as size, shape, color, etc. is out of reach of this paper and will not be discussed. Abstract syntax elements have one or multiple graphical roles. We generate automatically a part of the concrete syntax information by choosing graphical roles. The generated concrete syntax information is one “elected” possibility out of all different possible graphical representations.

This paper is structured as follows. Section 2 presents and discusses some existing works in the domain. Section 3 details the first contribution of this work i.e. the different classifications relevant for characterizing an abstract syntax element. Section 4 details the second expected contribution i.e. a semi-automatized process for concrete syntax generation before concluding about research perspectives.

2. State of the Art

This section discusses first the process of creating DSMLs, and then presents different frameworks, each one able to creating graphical DSML and to generate graphical editors.

In practice, designers are focusing on the DSML metamodel that defines the core concepts of the language. This phase is generally well controlled by practitioners, but is often separated from the design of graphical concrete syntax. DSMLs are better built in an incremental way, so an iterative approach is adopted that allows a validation and a verification process, by many language stake-holders and end-users. Such an iterative and incremental process would be possible only with an adapted tool support (Cho 2011). The workflow described in Figure 1, shows how to process temporally in parallel and with the same interest, the two aspects of the modeling process.

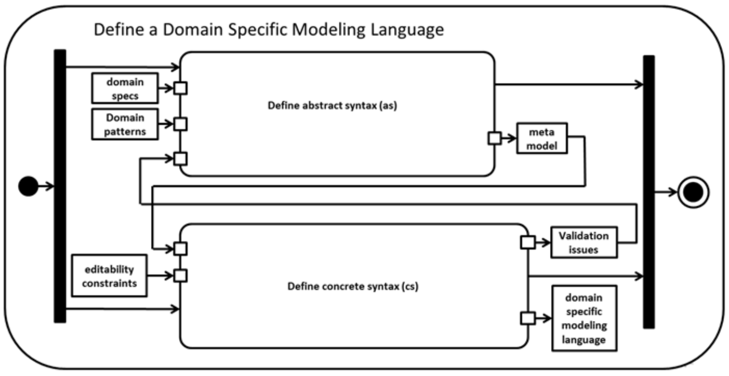
Define a Domain Specific Modeling Language
domain specs
Domain patterns
Define abstract syntax (as)
meta model
Validation issues
editability constraints
Define concrete syntax (cs)
domain specific modeling language

Figure 1. Define a DSML

2.1. Abstract syntax definition

An abstract syntax plays a central role in a language specification. It is the pivot between concrete syntax and semantics. The original meaning from the term abstract syntax comes from natural language, where it means the hidden, underlying, unifying structure of a number of sentences (Chomsky 1965). In computer science, particularly in MDE, an abstract syntax is represented by a metamodel. Meta-modeling languages such as the OMG's standard MOF provide the basic concepts and relationships in terms of which it is possible to describe a metamodel. Nowadays, multiple environments and metamodeling languages help to define an abstract syntax: Eclipse-EMF/Ecore (Steinberg et al. 2008), GME/MetaGME (Ledeczi et al. 2001), AMMA/KM3 (Jouault & Bézivin 2006) or XMF-Mosaic/Xcore (Clark et al. 2008). We use the graphical editor proposed by Eclipse-EMF, describing a metamodel using the EMF's metamodeling language Ecore. An example of a metamodel is shown in the Figure 2, which will be used as core element to our classifications and in a later phase as a source to the semi-automatic concrete syntax generator.

2.2. Concrete syntax definition

Concepts defined by the abstract syntax represent the underlying structure of a language. However, such concepts need additional information about their rendering to the end user. A concrete syntax of a language provides such information. We focus on graphical concrete syntax for DSMLs. The best graphical expressiveness of a graphical language depends on graphical economy of the representation. Moody in

(Moody 2009), presents a criteria for the expressive power of a graphical language. These criteria consider, first, the relative position of graphic symbols that make up the graphical language, and then, stylistic criteria about shapes, icons colors and line styles.

There are indeed several tools for creating a concrete syntax for a given abstract syntax and afterwards generating graphical editors. Graphical Modeling Framework (GMF) (Gronback 2009) is a framework based on a mapping between Ecore and Gef (a Graph drawing engine). This framework is powerful and widely used, but poorly documented. There are a number of frameworks based on top of GMF facilitating the process. TOPCASED propose a tool called “generator for graphical editors” allowing for a given Ecore model to define a graphical concrete syntax and then to generate the proper graphical editor (Pontisso & Chemouil 2006). Eugenia (Kolovos et al. 2010) is a framework based on top of GMF, annotating the metamodel with concrete syntax elements in order to generate the GMF artifacts. Diagraph (Pfister et al. 2013) is a tool for designing graphical DSML respecting the previously described process. The abstract syntax is represented by an Ecore metamodel and the concrete one is derived from the abstract syntax by a transformation targeting a generic metamodel of the graphical concrete syntax. Designing the concrete syntax implies the definition of this transformation; the latter is registered into the abstract syntax trough meta-data (annotations).

2.3. Correspondence mapping

The metamodel, shown on the Figure 2 contains the classes and the relations used to describe respectively the concepts and the relations of the personal computer hardware domain. To get an entire status of a language, this metamodel should be coupled with a textual or a graphical concrete notation. In our case, a concrete syntax should be defined by a set of graphical information e.g. type (node or edge), size, shape, etc. This information should be mapped to the abstract syntax elements. Such mapping is a bijection relationship between an abstract syntax element and a concrete syntax element. It is also called a correspondence mapping. Indeed, it is the correspondence mapping that determines how abstract syntax elements are going to be graphically represented, mapping them to a concrete syntax element (graphical information).

The work presented in this paper is demonstrated and implemented with Diagraph. We propose semi-automatically generated graphical concrete syntax for a given Ecore metamodel as a functionality included in this framework.

3. Classification

Understanding graphical domain specific modeling languages requires some definitions. We propose, in this section, three classifications of abstract syntax elements based on the graphical roles they have in a DSML and a general typology of graphical elements in DSML. The semi-automatic concrete-syntax generation process will be grounded on our typology.

As an illustration, we start from a toy language that describes a personal computer (PC) (see Figure 2). The whole collection of abstract syntax elements represent, together, the metamodel (the abstract syntax) of the language. Instances of those languages give rise to object graphs (graphs of objects which are instances of the classes contained in the abstract syntax). Such object graphs could be represented in a raw manner, as non-filtered and non-structured basic graphs, devoid of any cognitive power. Graphical modeling languages should provide specific visual representations, enabling their users to have good cognitive perceptions of the expressed statements. Thus, and this is a key concept, it is important to notice that every element of an abstract syntax (class or relationship) will not have a graphical representation. We discuss the ability of abstract syntax elements, to be graphically representable and/or represented. This is a shortcut to tell about the ability, for the instances of those abstract syntax elements, to be representable or represented. Among the elements that are not visible in graphical statements, some of them are not graphically *representable*, due to their role within the abstract syntax, and others are not graphically *represented* according to design decisions, generally for the reason of graphical economy (Moody 2009), or for distributing the language between different conceptual points of view which correspond, each, to a different graphical view.

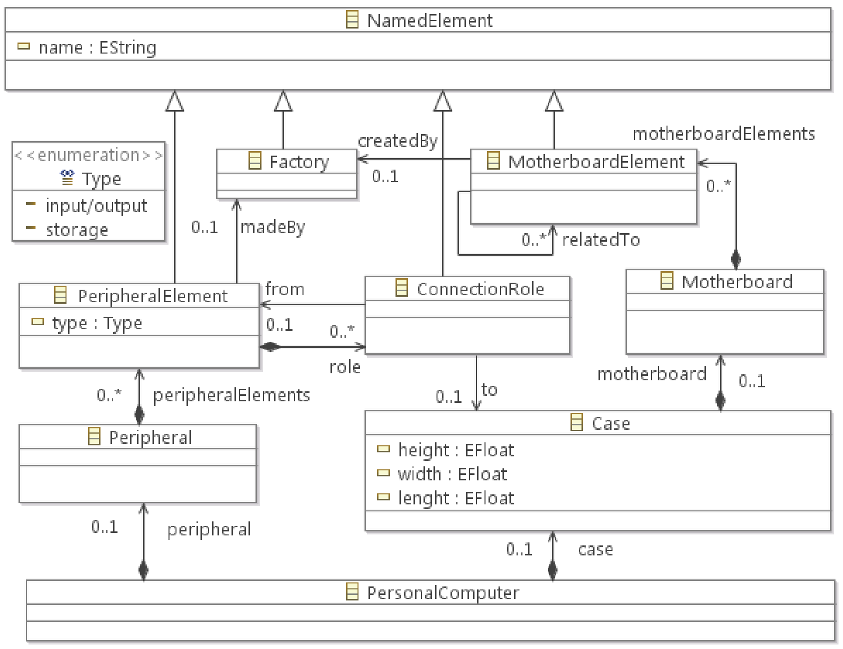

Figure 2. A metamodel describing a personal computer

Note that a graphical language may split the graphical space between several graphical views. Each view should be elected (chosen) by the language designer. As a starting point, we consider only one unique (structural) view for our PC language, which is related to the root class of the metamodel, also known as a *point of view (POV)*.

3.1. Graphical representability and Graphical representation

Depending on existing structures in the abstract syntax, the abstract syntax elements (class or relationship) can be classified into *graphically representable elements* and *graphically not representable elements*. Depending on design decisions (presented in a concrete syntax and a correspondence mapping), among the prior representable elements, we will distinguish, for a given graphical view, *graphically represented elements* and *graphically not represented elements*.

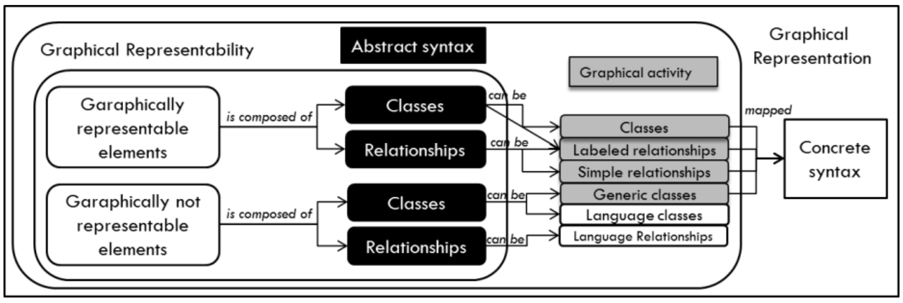

Figure 3. Graphical DSML elements classifications

3.1.1. Graphical representability

Based on the abstract syntax, we can distinguish *graphically representable and not representable classes* and *graphically representable and not representable relationships.* Figure 3 illustrates such classification. The left site of the figure represents graphically representable elements and graphically not representable elements. Each category is composed of graphically representable classes and relationships and graphically not representable classes and relationships. Together, representable elements (classes and relationships) and not representable elements (classes and relationships) form the abstract syntax as shown on the figure by black rectangular forms.

A graphically representable class is an abstract syntax element that is related by a relation of composition to another class, the latter having the role of the container, and if it is *connected* to the root class of the metamodel (the POV). The rest of the entities are identified as graphically not representable due to the inability of being instanced as a part of the language. We say that a class is connected to another class, if a set of navigable classes exists between the last two, e.g. a navigation that begins at the first class, passing through all classes, and finishing at the last. A graphically representable relationship connects two classes that are representable or subsume a representable class. Here also, the rest of the relationships are identified as graphically not representable. Table 1 represents Figure 2 graphically representable and not representable elements, considering *PersonalComputer* as a root class.

Table 1. Representable and not representable elements.

Representable classes	Not representable classes	Representable relationships	Not representable relationships
Peripheral, PeripheralElement, Case, Motherboard, ConnectionRole, MotherboardElement	Factory, NamedElement	peripheral, case, peripheralElements, role, relatedTo, from, motherboardElements, to, motherboard	createdBy, madeBy

3.1.2. Graphical representation

Based on a correspondence mapping, an abstract syntax and a concrete syntax, abstract syntax elements can be divided into *graphically represented elements* and *graphically not represented elements*. An abstract syntax element is graphically represented if, first, the element is a graphically representable element and, second, if it is mapped by the concrete syntax element (by a design decision). The rest of the abstract syntax elements, called *graphically not represented elements*, will not have a graphical representation, either because of a design decision or because they have a *graphically not representable* status.

Among graphically represented elements we distinguish, *graphically represented classes* and *graphically represented relationships* which are either *simple relationships* or *labeled relationships*. The common characteristic of these elements is that they are all, representable and mapped to the concrete syntax, while the difference lies in their underlying structures. A represented class is a single element inside an abstract syntax. For instance, if a conforming graphical information (concrete syntax element) is mapped to the class *Peripheral* (see Figure 2) the latter become a represented class. A simple relationship is also represented by a single element, a relationship (association or composition) inside an abstract syntax e.g. *motherboard*, and of course a conforming concrete syntax elements that is mapped to the prior. A labeled relationship is a compound structure, represented by a class and its attributes on the abstract syntax side, and by a labeled line on the concrete side, where the labels are mapped to the attributes. Such structure is often referred in the literature as a *class-association* pattern. For instance, the class *ConnectionRole* and the relationships *role*, *from* and *to*, are forming a labeled relationship inside the abstract syntax, between the classes *Peripheral* and *Case*.

Graphically not represented elements, if mapped by a conforming concrete syntax element, might have *indirect impact* of the graphical representation even besides the not representable status. Therefore, among the graphically not represented elements, we distinguish those that might have indirect graphical impact called *generic classes* (e.g. *NamedElement*) from those not having any possibility of graphical impact called *language classes* (e.g. *Factory*) and *language relationships* (e.g. *createdBy* and *madeBy*).

Generic classes should first, subsume (directly or indirectly) at least one represented class, second, they should hold a property (e.g. attribute) and third it should be mapped to the conforming concrete syntax element. This property is afterwards graphically inherited by the subsumed represented class. Note that the graphical inheritance is represented, in the abstract syntax as an inheritance relationship (direct subsuming) or a set of inheritance relationships (indirect subsuming) between the generic class and the represented class, and in the concrete syntax by a conforming concrete syntax element that is mapped to the property. It is the represented class that represents graphically the inherited graphical property of the generic class and thus the indirect graphical impact of the latter.

Language classes, besides the not representable status, do not subsume any represented class and thus the absence of any graphical impact. Such element has only cognitive and structural purpose in the abstract syntax.

As an illustration, Figure 3 describes a classification of graphical representations and the impact on a graphical concrete syntax.

The classification is illustrated on the left side of the figure while the right side of the figure illustrates a concrete syntax. In the middle (left), the abstract syntax elements are represented by black rectangular forms as representable classes, relationships, not representable classes and relationships. Each of these elements (representable and not) can be classified by the graphical representation classification based on their mapping on the concrete syntax. This is shown on the middle (right) side of the figure.

In order to factorize a graphical property from a generic class, the latter should be contained in the class by the mean of an attribute and/or a relationship (e.g. the attribute *name* from the generic class *NamedElement*). Since the factorization is of graphical nature, the concrete syntax mapping should be enriched with the proper information. Finally, a graphically represented class should be derived by inheritance from that generic class. Such a graphical inheritance mechanism will propagate, at the concrete syntax level, the attributes and the relationships that are inherited at the abstract syntax level. Therefore, the inheritance relationships relating a represented class to a generic class can be considered as *generic relationships*.

3.2. Graphical element activity classification

In the previous section we discussed a classification of graphical roles, distinguishing classes and relationships that are graphically represented or not represented, depending both on their role within an abstract syntax and on design decisions captured in a concrete syntax. If an abstract syntax element is graphically represented, it means that its graphical role is stored somewhere and associated to that element. All the graphical roles are stored within a concrete syntax model which acts as a layer of meta-data over the abstract syntax, therefore, a mapping should exists between the concrete and the abstract syntax. As we have seen above, there are also classes within the abstract syntax that are used to factorize graphically represented properties called *generic classes*. The inheritance mechanism that implements the factorization should also be noted into the concrete syntax which is mapped on the abstract syntax, in order to apply that design decision. On the other hand, there are classes that do not play any graphical role. This type of classes is called *language classes* and they do not figure within the concrete syntax layer. This classification aims to distinguish the last from the other elements, classifying them into *graphically active* and *graphically passive* elements. As illustration, in the middle (right) side of Figure 3, graphically represented and not represented elements are colored gray in order to represent the graphically active elements. The rest are considered as graphically passive elements.

A graphically active element is a language element that has associated information in a concrete syntax that causes the production of an element within the graphical notation of a domain specific modeling language. Such element is either a *graphically represented element* or a *graphically not represented generic class*. The graphically representable elements and generic classes of any abstract syntax are forming the graphically active elements.

A graphically passive element is a language element that has no associated information in a concrete syntax. Such element is also called an *abstract language element* (it could be a class or a relationship) and is used to hold a language information that is not graphically represented. The language classes and the not represented relationships of any abstract syntax are forming the graphically passive elements.

3.3. Graphical abstract syntax elements typology

It is very important to know what kind of elements can be presented by a graphical DSML and their different graphical representation possibilities. Therefore, this section defines a general typology for the representation of graphical DSML elements. Only graphically represented elements of the abstract syntax have an actual graphical representation (if they are mapped to a concrete syntax element), and therefore, this typology focuses only on graphically represented elements. Among them, we classified represented classes, and represented relationships, which can be either simple relationships or attributed relationships. When representing such elements within a graphical editor we refer to them as a *node, a simple link* and *a labeled link.* For a given (structural) view, we distinguish three different types of nodes, two different types of simple links and labeled links.

We classify the nodes into: *canvas*, *top level nodes,* and *child nodes*. The *canvas* is represented by the root class in the abstract syntax also called *the point of view (POV)*, containing the other classes and graphically represented as a screen (white board) where we add the rest of the nodes. The second type of nodes are the nodes that are disposed onto the *canvas* or *top level nodes*, while the rest of the nodes, that are embed into top level nodes or into other child nodes are called *child nodes* representing the third type.

Among the simple links we classify *relating links* and *embedding links*. A relating link is graphically represented as a *line* or an *arrow*, relating two nodes. In the abstract syntax it is represented by an association relationship between two classes. An *embedding link* is graphically represented as a *nested mechanism* between two nodes where one of them is container and the other is content. In the abstract syntax it is represented by a composition relationship between two classes (the nesting mechanism can start at the top level node, and the nesting depth is not limited; nodes are *top level nodes* at the first level into the *point of view*, and the deeper nested nodes are *child nodes*). In this case, there is neither line nor arrow, but the structure is, conceptually, an edge of a graph, graphically representing an embedding as a part of the view, the latter being mathematically a graph. This point is specific to our interpretation of diagramming, versus other approaches which consider that diagrams are unstructured drawings composed of graphical elements, related the one with the others but not necessarily as a graph structure, and (as we also do) mapped to a semantic artifact, the metamodel.

A *labeled link* is graphically represented as a *line* or an *arrow*, carrying one ore many labels and relating two nodes. The underlying structure of such label is represented by an attribute of a class and therefore the need of a supplementary class inside the abstract syntax.

4. Building the concrete syntax of a graphical DSML: needs and demonstration

The process of creating graphical DSML requires a language designer that is a double expert. First, an expertise in the domain he intends to model is required. Second, an expertise of creating “well-formed” DSML in the sense of language metamodeling is required.

A domain expert is a person who is an expert in a particular area or topic. This term is frequently used in the area of software development and the term refers to a particular domain (e.g. virtual machines, code generation, etc.) rather than a software domain. For instance, a “virtual machines” expert is a person that has a significant knowledge in the field of developing “virtual machines”. The domain expertise is an important prerequisite to the process of creating graphical DSML.

The expertise of creating “well-formed” DSML is on the one hand, in relationship with *metamodeling* and *modeling* (e.g. modeling using UML) and on the other hand, with small amount of concepts representing the *graphical language modeling*. Metamodeling and modeling are not the purpose of this paper and will not be discussed. For more details, see (Omg 2006; Mellor & Balcer 2002; Specification & Bars 2007). Graphical language modeling includes concepts that are specific for this kind of modeling. The expertise of graphical language modeling is knowledge of representing modeling concepts of any DSML i.e. the concepts introduced in element typology section. Together with general modeling, they form the expertise of creating "well-formed" DSML.

Definition: A "well-formed" DSML is a metamodel created by a domain expert following the concepts of metamodeling, modeling and graphical language modeling.

When defining a DSML, a language designer starts by defining an abstract syntax i.e. a metamodel. When the abstract syntax is incompletely defined, the designer starts defining how abstract syntax elements are graphically represented inside the concrete syntax, simultaneously creating the concrete syntax and the mapping of correspondences. The process is repeated until the abstract syntax is completely defined.

Each abstract syntax element has limited possibilities of graphical representation by a DSML which we refer to *graphical roles* or *roles*. For instance, if a class of an abstract syntax can be graphically represented; it can either be represented as a node, or as a labeled link porting an attribute. Hence, all graphical roles of each abstract syntax element can be calculated. Out of all calculated graphical roles, a set of conforming graphical roles can be chosen for each abstract syntax element. We call this the graphical role election process (see Figure 4). For each chosen role, the corresponding concrete syntax element can be generated and mapped to the corresponding abstract syntax element. Note that the graphical information representing shapes, size, colors, etc. is not generated and the generated concrete syntax information should be afterwards manually complemented by a language designer. Together, the abstract syntax, the generated concrete syntax elements and the mappings between them, form a DSML. A graphical editor can be generated for the latter. This is the basic principle of the semi-automatic concrete syntax generator. Such process is described by Figure 4, where the rounded rectangles represent processes and the normal ones represent data.

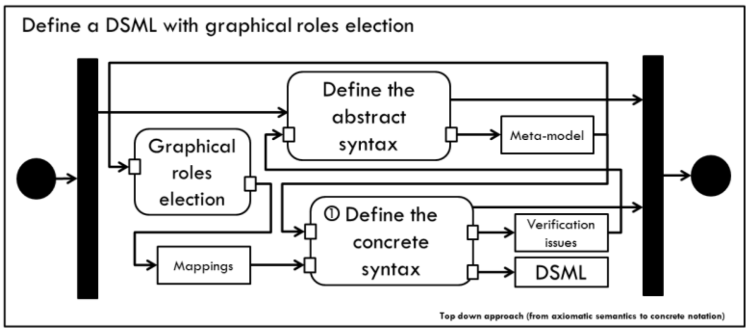
Define a DSML with graphical roles election
Define the abstract syntax
Meta-model
Graphical roles election
① Define the concrete syntax
Verification issues
Mappings
DSML
Top down approach (from axiomatic semantics to concrete notation)

Figure 4. Defining graphical DSML with graphical roles election

In this section, we detail the graphical role election process as main part of the semi-automatic concrete syntax generator, having as foundation the previously described classifications. The whole process is divided into four sub-processes.

First, a point of view election is applied electing a class that can represent most classes. Then the role marking process is applied. This process marks the possible graphical roles of each abstract syntax element. Afterwards, a metamodel search is applied, detecting a set of labeled relationships (class-association pattern). Elected roles of each labeled relationship class are filtered and roles representing labeled relationship are associated. Finally, the conforming concrete syntax elements are generated for each graphical role and then associated to each abstract syntax element.

4.1. POV election

The first part of the automatic concrete syntax generation process is the POV election. Each metamodel may have as many POVs as classes it contains. We choose as a language canvas, the POV containing most classes. Therefore, a metamodel search is first applied on the abstract syntax, in order to calculate the number of classes, each class can represent if elected as a POV. We apply such process on the metamodel described on Figure 2. The results are shown in Table 2. It is then clear why the class *PersonalComputer* is elected as the POV of the language.

Table 2. POV election result.

Point of view	Containing classes	Total containing
PersonalComputer	Peripheral, PeripheralElement, Case, ConnectionRole, Motherboard, MotherboardElement	6
Peripheral	PeripheralElement, ConnectionRole	2
PeripheralElement	ConnectionRole	1
Case	Motherboard, MotherboardElement	2
ConnectionRole	/	0
Motherboard	MotherboardElement	1
MotherboardElement	/	0
Factory	/	0
NamedElement	/	0

4.2. Role marking

The role marking process begins by a metamodel search, divides first graphically representable classes and relationships from graphically not representable classes and relationships. Each representable class is marked as represented class or node.

We do not mark representable relationships. They are used to supplement the classes that they relate with more graphical roles i.e. *container*, *content* or *source, target*. For instance, the composition relationship *peripheralElements* between the classes *Peripheral and PeripheralElement* will assign a graphical role of *container* to the class *Peripheral* and a graphical role of *content* to the class *PeripheralElement*. Such roles might afterwards be translated as a graphical embedding. Second, not representable classes are visited in order to distinguish generic classes from language classes. Here again, a metamodel search is applied verifying if a given not-representable class subsumes representable classes. In that case, the visited class is marked as a generic class.

We present this process on the metamodel shown by Figure 2. First, representable elements and not representable elements are marked as shown in Table 1. Afterwards, graphical roles are associated to each class as described by Table 3.

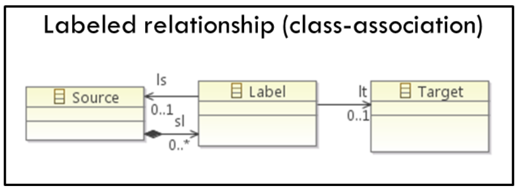

Figure5. Labeled relationship.

Table 3. Graphical roles election.

Class	Roles				
	Type	Source	Target	Container	Content
Peripheral	Node	no	no	peripheralElements	peripheral
Case	Node	no	no	motherboard	case
PeripheralElement	Node	no	from	role	peripheralElements
ConnectionRole	Node	to, from	no	no	role
Motherboard	Node	no	no	motherboardElements	motherboard
MotherboardElement	Node	related To	related To	no	motherboardElements
Factory	Languag e class	/	/	/	/
NamedElement	Generic class	no	no	no	no

4.3. Labeled relationships marking and role filtering

This phase consists of detecting labeled relationships (see Figure 5) of a given language. A metamodel search is affected on the metamodel identifying each possible class that can be represented as a label of such relationship (e.g. class Label of Figure 5). Such classes are always in relationship with two other classes, considered as source and target. The role marking phase has already assigned each class of the metamodel with graphical roles. Therefore, the roles associated to the labeled relationships elements should be regenerated taking into account this relationship. We continue the demonstration on the metamodel of Figure 2. Only the class *ConnectionRole* is detected as a label of a labeled relationship. The previously identified roles of this class (see Table 3) are deleted and a graphical role of labeled link is associated. The class *PeripheralElement* is filtered from the graphical roles target (*from* relationship) and container (*role* relationship), and assigned with the role source of the labeled relationship. The class *Case* is also filtered from the role target (*to* relationship) and assigned as target of the labeled relationship. Note that the rest of the graphical roles of a source and target classes of the labeled relationship are not filtered. For instance, the graphical role container (*motherboard* relationship) of the class *Case* is not filtered.

4.4. Automatic generation

Once all graphical roles are calculated and filtered, a set of conforming concrete syntax information can be generated, representing each calculated role. First, generic classes are complemented with graphical roles representing each property of the generic element. Finally, each graphical role is generated (concrete syntax) and associated to the corresponding class (correspondence mapping). Such process generates the concrete syntax information shown in Table 4 and Table 5 and maps it to corresponding abstract syntax elements.

Table4. Automatically generated concrete syntax (nodes).

POV (point of view)	*TLN* (top level nodes)	*Child nodes*
PersonalComputer	Peripheral, Case	PeripheralElement, Motherboard, MotherboardElement

Furthermore, the language designer should manually complete the concrete syntax and the mapping of correspondences, either by adding supplementary design decision, or by modifying the already auto-generated one. Finally, either a graphical DSML is created and a graphical editor can be generated, or a validation issue is detected which repeats the whole process (see Figure 1).

Table5. Automatically generated concrete syntax (links).

Embedding links	Relating links	Labeled links	Generic properties
Peripheral – PeripheralElement, Case – Motherboard, Motherboard – MotherboardElement,	MotherboardEleme nt – MotherboardEleme nt	PeripheralElem ent - Case	name

Figure 6 is an example of PC model conforming the PC language shown by Figure 2. The model is created by a graphical editor generated by the framework Diagraph.

5. Conclusion and perspectives

This paper has introduced and demonstrated a concrete syntaxes generator assistant for domain specific modeling languages. We demonstrated concepts and mechanisms allowing to guide and to assist a domain expert in order to define and formalize the concrete syntax of a graphical DSML. This work described how graphical DSML elements can be classified based on abstract syntax elements and on concrete syntax elements. A general typology for graphical DSML elements is described presenting the different elements that a graphical DSML can represent by a graphical editor. Furthermore, based on the classifications, we presented how a part of the concrete syntax can be generated automatically from an abstract syntax by a graphical role election.

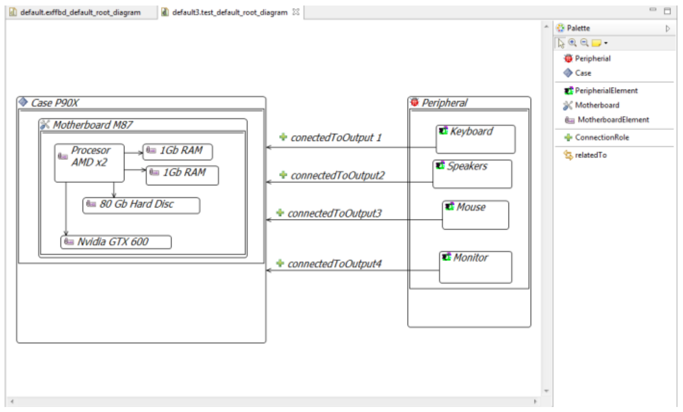

Figure 6. A model of PC conforming to the PC language.

This works open new perspectives. We aim to mathematically formalize the previously presented classifications. We also aim to provide a set of OCL constraints representing our classifications. Such constraints can guide the language designer in the process of creating the abstract syntax of a graphical language. We believe that such verification will improve the way of creating graphical DSML.

References

Cho, H., 2011. A demonstration-based approach for designing domain-specific modeling languages. ACM, pp. 51–54.

Chomsky, N., 1965. *Aspects of the Theory of Syntax* Anonymous, ed., MIT Press.

Clark, T., Sammut, P. & Willans, J., 2008. Applied metamodelling: a foundation for language driven development.

Fuentes-Fernández, L. & Vallecillo-Moreno, A., 2004. An Introduction to UML Profiles. *European Journal for the Informatics Professional*, V(2), pp.6–13.

Gronback, R.C., 2009. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, Addison-Wesley Professional.

Jouault, F. & Bézivin, J., 2006. KM3: a DSL for Metamodel Specification R. Gorrieri & H. Wehrheim, eds. *Lecture Notes in Computer Science*, 4037, pp.171–185.

Kent, S., 2002. Model Driven Engineering M. Butler, L. Petre, & K. Sere, eds. *Integrated Formal Methods*, 2335(2), pp.286–298.

Kleppe, A., 2007. A Language Description is More than a Metamodel. *Syntax*, (612), pp.1–9.

Kolovos, D.S. et al., 2010. Taming EMF and GMF Using Model Transformation. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*. Berlin, Heidelberg: Springer-Verlag, pp. 211–225.

Ledeczi, A. et al., 2001. The Generic Modeling Environment. In A. Ledeczi et al., eds. *Meta*. IEEE, pp. 1–14.

Mellor, S.J. & Balcer, M.J., 2002. *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley Professional.

MOF, O., 2002. OMG Meta Object Facility (MOF) Specification v1. 4. , (April).

Moody, D.L., 2009. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6), pp.756–779.

Omg, 2006. Meta Object Facility (MOF) Core Specification Omg, ed. *Management*, 080907(January), pp.1–76.

OMG, 2011. *UML 2.4.1-Infrastructure Specification*.

Pfister, F. et al., 2013. A light-weight annotation-based solution to design Domain Specific Graphical Modeling Languages. *ECMFA 2013, Montpellier, France, July 1-5, 2013.*.

Pontisso, N. & Chemouil, D., 2006. TOPCASED Combining Formal Methods with Model-Driven Engineering. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*.

Specification, O.M.G.A. & Bars, C., 2007. OMG Unified Modeling Language (OMG UML). *Language*, (November), pp.1 – 212.

Steinberg, D. et al., 2008. *EMF: Eclipse Modeling Framework* E. Gamma, L. Nackman, & John Wiegand, eds., Addison-Wesley Professional.