Harmonizing Textual and Graphical Visualizations of Domain Specific Models

Colin Atkinson
University of Mannheim
Mannheim, Germany
atkinson@informatik.uni-mannheim.de

Ralph Gerbig
University of Mannheim
Mannheim, Germany
gerbig@informatik.uni-mannheim.de

ABSTRACT

Domain-specific models, and the modeling languages that support them, have played a central role in the success of model-driven development and its ability to bridge the abstraction gap between software implementation technologies and human developers. However, the current generation of domain-specific modeling tools is firmly split into two camps — those that support textual domain-specific languages and those that support graphical domain-specific languages. Both camps have enjoyed significant success, but at the time of writing no mainstream tool supports both at the same time. This stops developers from swapping freely between textual and graphical visualizations of a given subject of interest, even if each form has clear advantages and/or disadvantages for different stakeholders. In this paper we present an environment which offers a solution to this problem by strictly separating concerns for notation (i.e. concrete syntax) and concerns for concepts (i.e. abstract syntax) and allowing them to be connected and mixed dynamically as required.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE); D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*

General Terms

Languages

Keywords

Multi-level modeling, domain-specific languages, orthogonal classification architecture, symbiotic languages, linguistic classification, ontological classification, modeling languages, diagram, visual languages

GMLD '13, Montpellier, France

http://dx.doi.org/10.1145/2489820.2489821.

1. INTRODUCTION

Domain-specific modeling has grown in importance over the last few years, and domain-specific modeling tools are now routinely used in industry. However, the market for domain-specific modeling tools has long been split into two groups — one group supporting graphical domain-specific modeling and one group supporting textual domain-specific modeling. Well known examples from the first group are MetaEdit+ [30], Graphical Modeling Environment (GMF) [14], Generic Modeling Environment (GME) [22], Poseidon for DSLs [13], Microsoft Visual Studio for DSLs [23] and ATOM3 [21], while well known examples of the second group are EMFText [15], XText [16], Spoofax [18], MetaDepth [20] and JetBrains MPS [17]. To our knowledge the intersection of these two groups is currently empty. In other words, there is no mainstream domain-specific modeling tool available today which supports both graphical and textual domain-specific modeling out-of-the-box.

For small, focused problems where the goal of the domain-specific modeling activity is to meet a very specific need of a single type of stakeholder this is usually not a problem. However, for larger, less-well-defined projects with multiple types of stakeholders this mutual-exclusion of textual and graphical visualizations of information can become very restrictive. The problem of mixing textual and graphical visualizations in a single editor has consequently received a lot of attention in recent years ([10], [12], [25], [27], [28]). When using state-of-the art commercial tools to develop mixed textual and graphical editors developers have to resort to workarounds that provide some way of transcending the different technology spaces and map graphically-viewable information into textually-viewable information. In the best case, the text-oriented and graphical-oriented tools are based on the same underlying infrastructure (e.g. XText and GMF are both based on the Eclipse Modeling Framework (EMF) [29] and the Eclipse Platform) and it is sufficient to develop glue code allowing the two editors to be used in tandem. In the worst case, when there is no common infrastructure behind the tools, a full suite of "model-to-text" and "text-to-model" transformations has to be defined (and subsequently maintained) to provide a full and coherent mapping of all the concepts in the languages driving the different tools. In all cases, supporting interoperability between the two kinds of visualization approaches is a tedious and often ad hoc task that invariably introduces accidental complexity in software engineering projects.

There are three fundamental roots to this problem. The first is the reliance of the majority of textual domain-specific

modeling tools on traditional parsing technology, based on context free grammar definitions, to define the structure of statements in textual languages, and the convention of entwining the definition of the abstract syntax with the definition of the concrete syntax. Thus, the majority of tools supporting textual domain-specific modeling languages work on the assumption that a dedicated abstract syntax is created to support the required concrete syntax, and if a different concrete syntax is required, a new dedicated abstract syntax is created for it. It is because this abstract syntax is concrete-syntax-oriented, and not problem domain oriented, that the underlying representation models for graphics-oriented tools and text-oriented tools are incompatible. Exceptions are tools like EMFText which try to solve the problem by annotating meta-models with information to generate parsers and textual editors. This means that languages must be built in such a way that they can be supported by the class of parsers supported by the tool with the consequence that the abstract syntax is influenced by the concrete syntax.

The second problem arises when using graphical and textual editors simultaneously while editing a model. Textual model editors based on parsers always create a new abstract syntax tree based on the currently parsed text. This model must then be merged with the model on which the graphical editor works. The merging problem is not trivial and cannot be solved in all cases, for example when identification information between the textual and graphical representation gets lost in the textual representation of a model. Such a loss leads to glitches when using graphical modeling editors and parser based textual modeling editors side-by-side.

The third problem is the reliance of all existing domain-specific modeling tools on a two-level physical modeling architecture to support classification. In other words, all editors in all the tools, no matter whether textual or graphical, basically support the creation and manipulation of instances of one fixed set of types. In other words, the types are hardwired into the code driving the editor while the instances of the types are "soft" (i.e. data). The consequence is that when the current generation of domain-specific modeling tools wants to "deploy" an additional layer in a domain-specific modeling language whose concepts are initially "soft" (and thus uninstantiable) they have to perform a major compilation and deployment operation to effectively create a new editor in which these formerly soft concepts are then hardwired. Again this introduces a lot of unnecessary accidental complexity and rigidity into the modeling process when employing more than one pair of classification levels in a domain model.

A fundamental solution to the problem of harmonizing textual and graphical visualizations of domain specific models must address all three of these problems. In this paper we introduce an approach for achieving this goal which addresses the first problem by introducing the notion of visualizers to drive all visualization processes and addresses the second problem by employing textual model editors based on projection technology of the kind supported by JetBrains MPS. This frees the tool from having to worry about the underlying parser generator's restrictions when defining the internal representation format. Furthermore, synchronization with a graphical editor is not based on any form of model merging because models are directly edited without the need for a generated parser. In other words, it makes this possible for text editors to utilize the same underlying model as graphics-oriented tools. However, the only mainstream tool currently supporting projectional editing for text, JetBrains MPS, only focuses on textual notations. The third problem is addressed by using a multi-level model environment to make all model elements and visualizers, at whatever level of classification, "soft" and thus amenable to change and manipulation at any time. By taking such an approach, the presented environment provides a uniform mechanism for supporting all possible forms of visualization, including tabular and wiki oriented forms of visualization, as well as textual and graphical forms. An additional benefit of the use of a multi-level modeling approach is that the DSLs can have a symbiotic relationship with the general-purpose notation used in the tool. This means that any of the domain-specific visualizations of a model-element is interchangeable with the general-purpose visualization at any time, at the touch of a button [2].

The remainder of this paper is structured as follows: in Section 2 multi-level modeling is introduced followed by a description of the domain-specific capabilities of multi-level modeling in the area of graphical and textual domain-specific languages (DSL) and symbiotic language support in section 3. Then, section 4 applies the approach to a small case study for modeling organizational structures. This language allows an organizational structure to be modeled in both a graphical and textual domain-specific language. The work closes with related work in section 5 and conclusions in Section 6.

2. MULTI-LEVEL MODELING

Multi-level modeling allows modelers to model a domain spanning an unlimited number of classification levels without resorting to awkward workarounds (e.g. UML Power Types or Profiles). This is made possible by the orthogonal classification architecture [4] which strictly separates ontological and linguistic classification into two orthogonal dimensions. The linguistic classification dimension represents the traditional model stack from a tool's point of view and describes how domain model-elements (L_1) representing domain concepts (L_0) are classified by the abstract syntax constructs of a single, ontological-level-spanning modeling language (L_2). The ontological classification dimension is a stack of ontological levels (a.k.a models) contained within, and orthogonal to, L_1. The sum of all ontological levels is called an ontology. Additionally, the concept of deep characterization is applied by giving each model element a linguistic attribute (a.k.a trait) called potency. This states how many levels the instantiation tree of a model-element can span and thus a model-element can influence. The potency of a model-element is either a non negative value or "*". Instances of a model-element have a potency one lower than the potency of their type. If the type has "*" potency, the instances can have an arbitrary value as their potency including "*". Model-elements with a potency of "0" cannot have instances. A potency is also attached to attributes (a.k.a durability) and their values (a.k.a mutability). The durability describes over how many instantiation steps an attribute can be handed over to instances and mutability over how many instantiation steps the value can be changed. In a modeling architecture featuring multiple levels it can happen that a model-element residing at the middle ontological levels is at the same time an instance of a type and a type for an instance. The term clabject is therefore introduced

to reflect the type (class) / instance (object) duality of an ontological model-element.

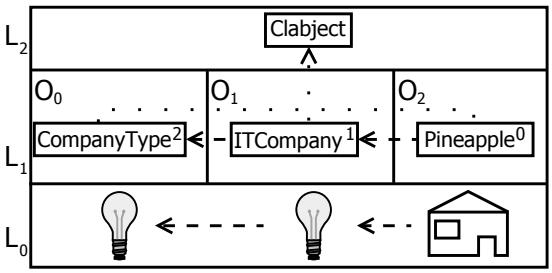

Figure 1: The orthogonal classification architecture.

Figure 1 shows an example of the orthogonal classification architecture and the concept of deep characterization. Vertically, a fixed number of three levels (L_2 - L_0) exist describing the multi-level modeling language at L_2, the ontology at L_1 and the real world at L_0. The ontological levels horizontally contained in L_1 show an example in the domain of an organization modeling language. The example spans only three ontological levels (O_0 - O_2) for space reasons. All three ontological model-elements (CompanyType, ITCompany, Pineapple) are instance of the linguistic metamodel-element Clabject (dotted vertical arrows) stating that these model-elements can be types and instances at the same time. In this example only ITCompany is simultaneously a type and an instance because it is the only model-element on a level that is both classified by a level and classifies another level. Ontological instantiation is indicated through horizontal dashed arrows. Each model-element has a potency as superscript next to its name. CompanyType for example has potency 2. The instances of CompanyType, here ITCompany, have a potency that is one lower than the potency of their type, CompanyType, resulting in a potency of one. At the lowest level an ITCompany named Pineapple resides with a potency of zero because it is created from CompanyType through two instantiation steps each lowering the potency by one. The rendering shown here uses the Level-agnostic modeling language (LML), the general-purpose language for rendering multi-level models. A modeler, however, can also define domain-specific renderings for model-elements as described in the following section.

3. SYSTEMATICALLY SEPARATING ABSTRACT AND CONCRETE SYNTAX

At the present time our multi-level modeling environment, MelanEE [1], supports graphical and textual visualizations of domain-specific languages but tabular and wiki-oriented visualizations are also planned in the future. All visualization forms are realized in a uniform way using the concept of visualizers. These allow the representation of model-elements to be defined and updated in a highly flexible way, allowing the rapid prototyping and changing of domain-specific languages. In this section we introduce the concepts of visualizers and explain how they are realized in MelanEE.

3.1 The Visualizer Concept

As its name implies, a visualizer determines how a model-element is visually represented. Two general kinds of visualizers are currently supported — Level-agnostic Modeling Language (LML) visualizers and DSL Visualizers. LML visualizers define the general-purpose visualization of clabjects and their instances / subclasses in the LML. Their attributes store all the values needed to define how the visualization symbol should appear, such as how each trait is displayed. The default visualization, as defined by the LML, is indicated by the value "default". For example, by default the potency of an attribute (a.k.a. durability) is not shown if it has the same value as that of its clabject. This default behavior can be overridden by changing the value of the attribute to "noshow", "show" or "tvs". The first of these values, "noshow", indicates that the value should never be shown no matter in which circumstances. This value is often used to hide the level a clabject is located at, because this is unimportant information in most cases. The second value "show" specifies that a trait should always be displayed. The durability and mutability of an attribute are often set to "show" in educational models to make their values explicit. The last value "tvs" is used to display the value in a special location under a clabject's designator (containing besides other things the name of a clabject), the trait value specification. In addition to these attributes, a visualizer contains information about what should be displayed in a clabject's designator and whether a domain-specific rendering is desired. The designator trait of a visualizer indicates what to display in the header compartment of the clabject's general-purpose rendering — the LML —, including whether to display information about its heritage, classification and location. More information on the LML can be found in [5].

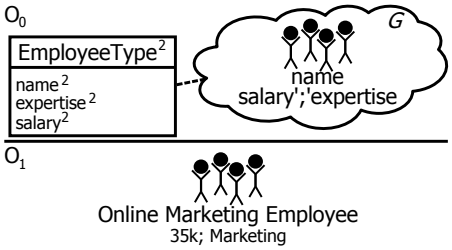

Figure 2: The definition of a graphical domain-specific rendering.

Domain-specific visualizers can be used to configure domain-specific visualizations of clabjects and their instances / subclasses. More specifically in case of a graphical visualization, they allow various kinds of graphical symbols to be associated with clabjects based on the concepts of layouts, geometric shapes and labels — an approach to graphical modeling language definition that has been widely implemented (e.g. MetaEdit+, GMF-Tooling, OMG Diagram Definition [24]). The basic idea behind this approach is shown in Figure 2 which shows a small excerpt of a more complete DSL definition shown in Figure 7. The example focuses on the definition of a graphical notation for one model-element, EmployeeType, which has three attributes, name, expertise and salary. In the domain-specific visualization these three attributes should be shown underneath a graphical symbol resembling the notion of EmployeeType. In the figure, the attachment of a symbol to a clabject is represented by a cloud, but in a tool such as MelanEE this is achieved by creating a visualizer model-element. Here, a scalable vector graphics (SVG) figure is placed in the first row of a one column table layout. The label mappings for the attributes

to be displayed are placed in the second row of the table layout.

Since there can be several visualizers attached to a clabject, a user can decide on-the-fly whether to work with the general-purpose or one of the domain-specific notations of a model-element. This can be extremely useful to domain novices in cases where different metaphors in a modeling language are quite similar to each other. We refer to such a relationship between a domain-specific and a general-purpose language (i.e. where the symbols defined by one can be intermingled on a clabject-by-clabject basis with the different symbols defined by another) as a symbiotic relationship.

Textual domain-specific languages are defined using the concept of visualizers, too. In our multi-level modeling tool, MelanEE, only a simple language is need for this purpose. This is possible because most of the structural information such as multiplicities etc. is defined in the model itself and not the concrete syntax definition. On the contrary, in EBNF-focused textual domain-specific language tools such as XText the concrete and abstract syntax definitions are mixed together. MelanEE implements textual domain specific language definitions in a similar way to EMFText which annotates a meta-model with information for textual rendering. The concept of visualizers is similar to the concept of annotating an Ecore model with textual rendering information. For each model-element a visualizer containing the definition of the text snippet for rendering is attached. From this information an editor for editing the model in a textual language is generated.

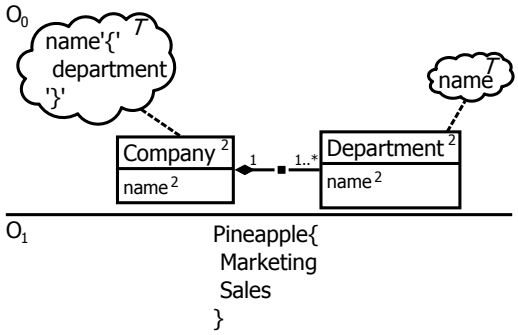

Figure 3: The definition of a textual domain-specific visualization.

The language used for defining textual syntax revolves around two concepts and has thus a very low complexity. One is the definition of parts of the textual language which cannot be edited, the other part is a mapping of text to attributes and references in the model which are editable in the resulting editor. An example of a textual language definition is displayed in Figure 3 which again is an excerpt of the organization modeling language in full detail shown in Figure 7. This language definition excerpt defines two model-elements, Company and Department. A textual visualizer represented as a cloud is attached to each one of these concepts. The visualizer of Company states that first the name of a company is displayed followed by its Departments enclosed in curly brackets, while the visualizer of Department states that a department is visualized by the value of its name attribute. O_1 shows an instance of the modeling language represented by the instances' textual notation. A company called Pineapple has been instantiated containing two departments called Marketing and Sales.

3.2 The Visualization Search Algorithm

The idea of annotating model-elements with rendering information used in an editor, and applying a search algorithm to find what symbols to render is not new to meta-modeling. For instance in [11] Ecore meta-models are annotated with information for textual visualization and a search algorithm is applied to find the most appropriate visualization information for a model-element. That algorithm, however, does not cover multi-level modeling scenarios. A multi-level visualization search algorithm such as that described in [4] (and steadily refined since then) searches up a clabject's complete stack of classifying levels as well as its own level. In general the visualization search algorithm first searches in the inheritance hierarchy of a clabject for rendering information. If no visualization information is found the next level in the classification hierarchy is searched and so on until all the classification levels above that of the clabject to be visualized are searched through. This is done recursively over all classifying levels until a result is found. If the algorithm ends without finding visualization information at the ontological levels, the visualization of the model-element's linguistic type, which is the LML rendering, is used. In fact, because of MelanEE's support for symbiotic languages (i.e. toggling the rendering of a clabject between DSL or GPL notation at will) a modeler can switch off this visualization search algorithm at any time, for any individual model-element, and immediately use the GPL visualization. The visualization search algorithm is not limited to graphical visualizers, but applies to any kind of visualizer including textual and tabular visualizers. For illustration reasons the application of the algorithm is only shown for graphical visualizations in Figure 4. However,it works in the same way for textual and tabular visualizations etc.

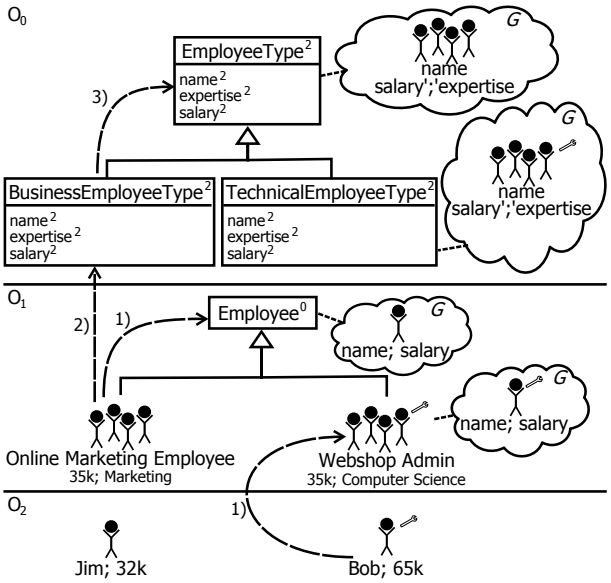

Figure 4: The visualizer search algorithm. Dashed lines indicate search order of algorithm.

Figure 4 shows the application of the visualization search algorithm in the context of the company modeling language introduced previously. EmployeeType, residing on level O_0,

is modeled with a default graphical visualization representing a group of employees. This default visualization is overridden by TechnicalEmployeeType which shows a wrench in addition to the group of employees. BusinessEmployeeType and its instances on the contrary are represented by the visualization defined for EmployeeType. On level O_1 the EmployeeTypes are instantiated by different kinds of employees. An abstract base class, Employee, which has a default employee metaphor attached, is introduced as root model-element. To prevent its subtypes to use this visualization information instead of their type's it is marked as to instances applicable only. This not indicated in the figure for overview reasons. The attached visualizer accesses two attributes which are not present in the Employee model-element. This is allowed by the visualization algorithm as non existing mappings are simply ignored during run-time offering a high degree of flexibility when defining visualizers within an inheritance hierarchy. Online Marketing Employee is instantiated as instance of BusinessEmployeeType and Webshop Admin is instantiated from TechnicalEmployeeType. The visualization search algorithm is applied to find a suitable visualization for each model-element at O_1. Employee does neither have supertypes nor ontological types, but has a graphical visualizer attached to itself. Thus its own visualizer which is found by the search algorithm can be used for visualization. In the example, however, it was decided to use the GPL visualization for the model-element and not to apply the result of the visualization search algorithm. Online Marketing Employee and Webshop Admin are both rendered using their domain-specific notation. Both do have a common supertype, Employee, which stores a graphical visualizer. The visualizer is marked as only applicable to instances which is omitted in the figure. Thus no visualization information is found in the inheritance hierarchy. Hence, the visualizer search algorithm continues searching at the next ontological type level. In case of Webshop Admin a graphical DSL visualizer is found at its ontological type, TechnicalEmployeeType. A visualizer for Online Marketing Employee cannot be found at its ontological type, BusinessEmployeeType. Hence, the inheritance hierarchy of BusinessEmployeeType is searched which results in a graphical visualizer found at EmployeeType. Level O_2 contains two employees, Jim an Online Marketing Employee and Bob a Webshop Admin. For both the visualizer search algorithm is applied in the same way as described previously for the Online Marketing Employee and Webshop Admin which results in a visualization for Bob gained through its ontological type Webshop Admin and a visual representation for Jim gained through the supertype of its ontological type Online Marketing Employee namely Employee.

3.3 Graphical DSL Editing

Graphical domain-specific modeling visualizations are supported in the same model editor as general-purpose visualizations without the need for any recompilations or redeployment. A modeler can change between a clabject's general-purpose or domain-specific rendering at the click of a mouse. All types available for instantiation in a domain-specific language are displayed alongside the general-purpose language elements in a tool palette on the right hand side of the editor. The properties view at the bottom displays the attributes to be edited of the currently selected model-element. A modeler can choose to filter the information displayed while using a domain-specific language by for example hiding all ontological levels except the one he is working with and hiding all the general-purpose language modeling tools from the palette. The properties view also allows a domain-specific view on model-elements which only displays ontological attributes and hides linguistic attributes which are irrelevant when using a domain-specific language. Through these mechanisms a modeler can customize the degree of domain-specificity of features offered when using the tool. A user more confident within a domain can work fully with the domain-specific features, whereas a language novice can choose to include general-purpose features as well.

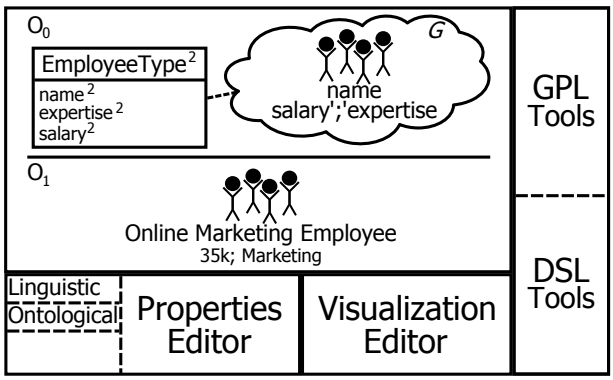

Figure 5: Mock-up of the graphical DSL editor.

A mockup illustration of the MelanEE editor is shown in Figure 5. The main modeling space is positioned at the central, left part of the screen consuming the most space. The editing area shows two levels, O_0 and O_1. A user can, however, decide which levels to work with as mentioned earlier. At the left bottom the properties view, with two tabs on its left hand side, is located. This view is used to change the values of linguistic traits and ontological attributes. The first tab contains the Linguistic properties page which displays linguistic traits of the currently selected model-element, such as the potency, and can be used to change these values. The second tab contains the Ontological properties page which displays the ontological attributes of the currently selected model-element and is thus most frequently used in the DSL view. It is needed because in most DSLs a user cannot select an ontological attribute within a model-element's symbol and change its linguistic value trait. To do so all ontological attributes of the currently selected model-element and their value traits are made available for editing in this view. Next to the properties view, on its right side, a view for editing the visualization information of the currently selected model-element is placed. This view does support the definition for graphical, textual etc. editors based on the concept of modeling visualizers which are available for these different view kinds. The right side of the environment shows the tool palette which is divided into two parts, the GPL Tools and DSL Tools part. The GPL Tools palette defines the general LML types available for instantiation, while the DSL Tools palette offers the ontological model-elements available for instantiation. This palette responds in a context sensitive way. After a user has selected the ontological level / container into which one wishes to place a model-element, the DSL Tools palette offers all the ontological types that can be instantiated at this place.

3.4 Textual DSL Editing

Two basic ways of editing models using a textual concrete syntax exist — one based on parsers and the other based on projection. The difference between the two is how they build up and edit the model representation (i.e. abstract syntax tree) of the edited text. The most wide spread approach, used by technologies like XText and EMFText, uses a parser driven by some kind of context free grammar specification (e.g. defined in an EBNF dialect). This approach, however, has a few disadvantages. A general problem, not related to the combination with model-driven technologies, is that a parser generated from a grammar is always limited to the parser class supported by the parser generators used in the employed language workbench. This requires a language engineer to know which parser class is required to parse a certain language before choosing a tool to implement a textual domain-specific language. In addition, when designing a domain-specific language, an engineer has to keep in mind the capabilities of the underlying parser generator. In some cases it can happen that constructs in a grammar need to be expressed in a certain way to work with a generated parser while other ways of expressing the same language construct do not work.

Also problems arise when editing models through text which are used in more than one editor at the same time (e.g. in a graphical and textual model editor). Graphical editors for example often store meta-information about the layout of a model which needs to be preserved while editing a model. Here problems occur when saving the content generated by a parser. Parsers cause problems because they always create a new model (i.e. abstract syntax tree) out of unstructured data (i.e. plain text). This model then needs to be combined with an existing model in order to keep meta-information like IDs, layout information etc. defined in the edited model.

Merging algorithms are often applied to merge models generated by a parser with the existing edited model to preserve in it defined meta-information, such as layout information. Such merging algorithms rely on identification information which must be present in both the textual model representation and the edited model itself. In cases in which this information gets corrupted, the merging algorithm fails to merge the models. This failure is compensated for by removing the model elements from the edited model which have no matching identification information in the edited model and the textual representation. Afterwards, a new model element is created for those model elements present in the textual syntax which do not have a counterpart in the edited model anymore. In this way unmatched model elements in the edited model are replaced with new model elements containing the same information as the previously deleted ones. This behavior leads to a loss of meta-information (e.g. for layout) about model elements because for the newly created model elements this information is not present. Losing layout meta-information in a graphical editor often leads to glitches like model elements getting repositioned at an arbitrary position after transferring a change from the textual model editor to the edited model. A user then needs to manually layout the corrupted layout of the model. An additional problem when editing a model through text arises with statements which do not exactly conform to a language's textual syntax definition. In these cases it can happen that a textual model editor either prohibits the saving of a textual model or starts deleting the elements (and all of their children) that do not conform to the concrete syntax. This again can cause a loss of information in an edited model. Strictly speaking, when a user changes the keyword "class" to "clas" in java, for example, the whole class including its content should be deleted from the saved model. The approach of prohibiting a save in cases of invalid concrete syntax is for example used by the Eclipse Project's OCLInEcore editor [9], which allows Ecore models to be edited using a textual editor. The textual editor prevents the user from saving the diagram until the textual representation of the model is in full conformance with the textual syntax definition. This is, however, not very practical because it prevents a user from saving his / her work while editing a model. Moreover after each save the parsed text and the model are merged. If an ID gets corrupted while editing the model's textual representation it can happen that no model-element in the edited Ecore model can be found for the text snippet currently edited. This causes the old model-element be deleted and a new one to be created. As mentioned earlier, this is not a problem as long as this model is not edited with a second editor that stores meta-information for a model-element such as a graphical model editor. Otherwise the previously described glitches when editing the model in the graphical and textual model editor side-by-side can occur in this case.

The second way to implement a textual model editor is to use projection. An example of a textual textual language workbench based on projection is JetBrains MPS. Projection provides a continuous connection between the model and the text representing the model. Text is no longer parsed but changes are directly written into the model. This direct editing of the abstract syntax tree makes parsers obsolete and frees languages from the constraints of the parser classes that can be generated by a certain textual language engineering tool. Also the model merging step is not required and thus, no meta-information (e.g. layout) is lost in cases where re-parsing would cause a loss of such information. The drawback of such an approach is that it cannot deal with unstructured data "out of the box". In contrast to a parser based editor one cannot paste text into a projectional editor because it is not capable of transforming this text into a model. This weakness could be reduced by combing a projectional editor with a generated parser that allows existing text to be imported. Such an import mechanism is, however, again limited to the power of the parser generator. The textual multi-level model editor in MelanEE is based on a projectional approach.

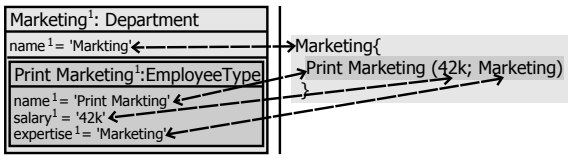

Figure 6: Textual model editor and weaving links pointing to abstract syntax elements.

Figure 6 shows how projectional editing is realized in our Multi-level modeling tool, MelanEE. The left hand side shows the general-purpose graphical visualization of a model-element in LML and the right hand side shows a textual representation of the same information. Strictly speaking the left

side is a representation of the model's abstract syntax in memory which could also be displayed as a graphical model side-by-side with the textual editor as indicated in the figure. The dashed arrows between the text and the model indicate the connections that are established and maintained by the projectional text editor. The connections are realized by a bi-directional weaving model that connects the model and its textual representation. This weaving model looks very much like the concrete syntax model of the text displayed in the text editor with one exception. This exception is that the concrete syntax model is expanded with weaving links to the edited model. If a user changes the text in the text editor, the weaving model (the concrete syntax model) is immediately changed. This model is monitored for changes by the model stored in memory which, on the event of a change, updates its model-elements which are connected to the changed concrete syntax model elements. The same mechanism also works in the other direction. As soon as a modeler changes the model in the memory (e.g. through a graphical editor) the concrete syntax model of the textual representation is updated which in turn updates the text which is represented by this model. Through this mechanism it is always guaranteed that the multi-level model and its textual representation are synchronized.

The model editor ensures that a user can only make changes which correspond to the textual concrete syntax definition of the model. This prevents situations in which a user can not save a model as this would cause an inconsistency between concrete syntax and edited text and would thus cause an accidental loss of information during a save operation. To achieve the target of only allowing textual representations which conform to the textual syntax definition of a model the idea of syntax directed editing is used to realize all aspects of the model editor. Keywords are marked with a grey background when selected and cannot be changed. Only delete operations are allowed on keywords. If they are optional they can be deleted without any effect on the model. In cases where they are not optional the user is asked whether to delete the whole model element and all model elements it contains. Literals which only consist of white-spaces can be extended by white-space or parts of them can be deleted for formatting reasons. New model elements can only be created by invoking the context sensitive and language driven features of the editor. Attributes can be edited like usual text. Validation errors introduced through editing are displayed as editor annotations in the textual editor. These errors cannot only be fixed in the textual editor but also in others, such as graphical editors, which then fix the error in the textual view by remaining synchronized with it.

4. MULTI DOMAIN-SPECIFIC VISUALIZATION MODELING

In the previous section the visualization and editing mechanisms for textual and graphical domain-specific multi-level model editing have been presented in isolation. The unique characteristic of MelanEE's domain-specific language support is that the defined editors can also be used together in an integrated manner to facilitate the viewing and editing of a model in multiple visualization forms (e.g. graphical and textual) at the same time. To do so neither separate tools using different technologies nor glue code that connects two modeling technologies within one environment needs to be developed. A modeler can use different visualization forms side-by-side by simply defining visualizers for the visualization form to be used and open the model in different editors at the same time. Figure 7 shows the editing of a model using two different domain-specific language visualization forms in the context of the earlier partly introduced organization structure modeling DSL. This example shows one model being edited using two separate editors for the same model. It is also planned to support the mixing of visualization forms within editors. A user can for example host a textual or tree based editor within a graphical editor to improve the overall user experience. An example where hosting a textual editor within a graphical one is a significant advantage is found in the domain of UML Class diagrams. These diagrams can use graphics to display classes and a textual editor hosted within the graphical editor to offer convenient attribute editing with features like coding assistance.

The top of Figure 7, O_0, shows the language's meta-model with the definition of the textual and graphical concrete syntax defined side-by-side. To indicate the definition of a DSL visualizer the notation of a cloud attached to the model-element for which the concrete syntax is defined is used. The graphical definition is indicated by a *G* and the textual definition by a *T* in the upper right of the cloud representing the visualizer. The language defined at the top provides the possibility to model Companies consisting of Departments which again consist of different EmployeeTypes. A Company is represented by the visual metaphor of a house containing Departments represented by dashed rectangles containing EmployeeTypes rendered through a group of stick men. From a textual view point a Company is rendered by first printing its name, followed by its Departments in curly brackets which are rendered by their name and contain their different types of employees in curly brackets. These different EmployeeTypes are visualized by stating their name followed by their salary and expertise separated by a semi-colon and enclosed in round brackets. An instantiation of this organization modeling DSL with a fictive IT company called Pineapple is shown at the bottom level of Figure 7. The company consists of two departments, a Marketing and Sales department. These departments employ different types of employees, with different salaries and different skills, e.g. Webshop Admins with a salary of 45k and skills in Computer Science. The model is shown in a graphical DSL (left side) and textual DSL (right side) side-by-side in the figure. A language user, however, has different options when editing this model — a) only showing the graphical visualization of the DSL, b) only showing the textual visualization of the DSL and c) showing both graphical and textual forms side-by-side. The option shown in the figure is the concurrent editing of the model in its textual and graphical notation. A user can create the content using the projectional, graphical layout preserving, textual DSL editor as this is assumed to be faster than using a graphical editor. On the other hand the user can at the same time manipulate the model from a graphical point of view, for example changing the layout of the graphical diagram, without the need to switch modeling tools. This enables a user to exploit the strengths of multiple domain-specific visualization forms, here textual and graphical, at same time. In addition, one could also get offered a third view on the same model such as a tree based model editor focusing on the containment hierarchy of the model. Such a view again integrates seamlessly into

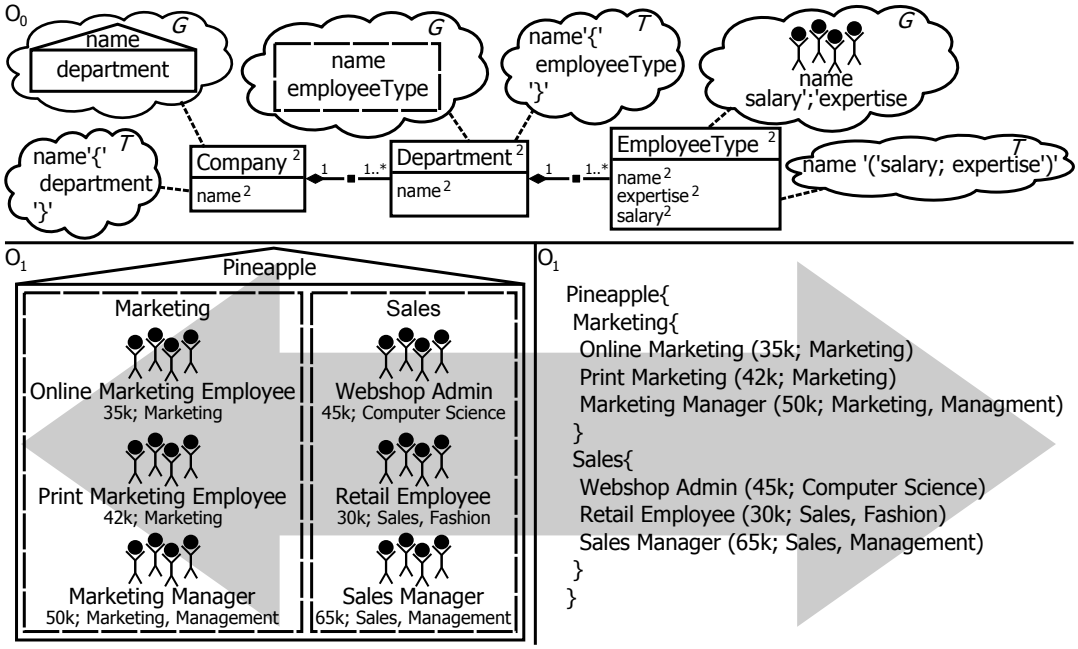

Figure 7: Simultaneous graphical and textual editing of a model.

the modeling language editing experience. One could even imagine a third level which can contain instances of the types modeling a company in details (e.g. having different Marketing departments with different employees in different roles such as Marketing Manager). It would also be possible to define a second textual interchange syntax on the same model which can be used for exchanging content modeled using the graphical and textual multi-level DSL with other tools.

5. RELATED WORK

A lot of work as been done in the area of combining textual and graphical languages with each other. The work discussed here mostly focuses on combining two tools with each other. Some focus on integrating XText with GMF based model editors by writing glue code between the two by hand. These scenarios support viewing a whole model in code and graphical notation side-by-side or embedding textual editors within a graphical one gluing together two different technical stacks (e.g. XText and GMF in [12]) and synchronizing the textual and graphical representations on save operations. There are also approaches that work on two separate models and apply a merge algorithm at run-time independent of load/save mechanisms and synchronize views based on these separate models [28]. Other work describes a framework for augmenting a graphical editor with context menus for editing parts of a model using a textual view [27]. All of these techniques however involve the application of multiple technologies to define a single modeling environment. No uniform technology and modeling concepts are used to create both a graphical and textual editor. Furthermore, the first two technologies require complex glue code to be written in order to connect a graphical and textual editor. The integration of textual and graphical languages has also been implemented in some tools e.g. UML Activity Modeling [10] or in the context of the KIELER tool [28]. In [25] Atom^3 is extended to include textual views on an underlying model. The textual views are defined on a metamodel and grammar based parsers are generated to parse the textual model instance and write it back to the underlying model of the whole system. The authors, however, make no statement about the editing experience when working on the same part of a model using a graphical and textual notation at the same time. They do mention that the approach currently has some shortcomings such as the fact that the only direction that is currently supported is text-to-model.

In the area of multi-level modeling not much work has been done on offering graphical and textual editors side by side on a model. MetaDepth [8], a multi-level modeling tool based on textual syntax, makes it possible to view a model in a read-only graphical view. This view does not support domain-specific notations which can be defined by a language engineer. Domain-specific textual languages can be created by using this tool [20].

The work here is not related to view-based software modeling methods which differ in the sense that a whole system is modeled using views. These views often do not overlap each other or even describe the same aspects of the underlying system using two or more different visual notations. Examples of such view based modeling approaches are enterprise modeling frameworks like Archimate [19] or ARIS [26] and their supporting tools. View based modeling is also applied in software engineering for the purpose of managing the complexity of refactorings [7] or developing component based software systems [6]. In contrast the work presented here focuses on the definition and usage of multiple concrete syntax (e.g. graphical and textual) in an integrated and uniform way for one single model and one single purpose. Hence, the technology here would be more suitable to define views of a view based tool than realizing the tool with all its transformations etc. itself. In [3] MelanEE's domain-specific modeling capabilities are used to realized the views of such a view based modeling tool.

6. CONCLUSIONS

In this paper we identified the current fragmentation of domain-specific modeling tools into two groups — one focusing on the textual visualization of models and the other on the graphical visualization of models, and have explained why this is suboptimal. We then explained the underlying reasons for this problem and presented a fundamental strategy for overcoming them. Finally we described how our prototype modeling environment, MelanEE, applies these strategies to unify and harmonize the way model content is visualized. The key idea is to use visualizers to fully disconnect concrete syntax concerns from domain information representation concerns so that the former has absolutely no influence on the latter. This allows multiple kinds of representation formats (corresponding to the different kinds of visualizers) to be applied to the same model context at the same time, and even mixed so that it is possible to incorporate a textual rendering of a part of a model within a graphical rendering of another part of a model. Tabular and Wiki oriented visualizations can also be supported using this approach by adding the necessary visualizers. In fact, in the long run even the GUI of a system (i.e. the GUI widgets) could potentially be supported using this approach.

A natural and profound consequence of this technology is that the notion of a language should be completely decoupled from the notion of its concrete visualization format. Thus, a language should no longer be characterized as being textual, graphical or tabular, since these are not properties of the language per se but of visualizations of a language. In other words, we propose that a language is characterized by the domain concepts it provides support for (i.e. a business process modeling language provides support for task and decision points, while a state modeling language provides the notions of states and transitions etc.) rather than by a particular concrete visualization that may have been associated with it "out of the box". It follows that using this technology a given domain-specific modeling language can have several visualization formats, some of them textual, some of them graphical.

This terminology is consistent with an ongoing trend for general-purpose modeling languages where the essence of what the "language" is has moved away from concrete syntax towards the meta-modeling concepts (c.f. abstract syntax). Such a trend has been very pronounced with the UML. However, the terminological consequences of this trend with the UML have not been carried through to its logical conclusion, so that many people still associate the Unified Modeling Language with the original concrete symbols rather than with its meta-model's concepts. Ideally, the concrete visualization should be regarded as one possible visualization of the UML (albeit the default one). Other textual visualizations have existed for a long time (e.g. XMI, HUTN).

To our knowledge MelanEE is the only prototype tool that supports textual and graphical visualizations of DSLs side-by-side in a uniform way by combining the principles of multi-level modeling with projectional editing (in the case of textual visualizations). However, we hope that when the benefits become apparent other prototypes will follow. Our group plans to continue this research by adding examples of other classes of visualizers in due course, such as tabular and wiki visualizers. This enriches the number of editors in which a modeler can view parts or the whole model in a unified way without dealing with multiple domain-specific language engineering tools. We also plan to expand the range of DSLs available in MelanEE.

7. REFERENCES

[1] C. Atkinson and R. Gerbig. Melanie: multi-level modeling and ontology engineering environment. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, pages 7:1–7:2, New York, NY, USA, 2012. ACM.

[2] C. Atkinson, R. Gerbig, and B. Kennel. Symbiotic general-purpose and domain-specific languages. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1269–1272, Piscataway, NJ, USA, 2012. IEEE Press.

[3] C. Atkinson, R. Gerbig, and C. Tunjic. A multi-level modeling environment for sum-based software engineering. In *1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO) 2013*, 2013.

[4] C. Atkinson, M. Gutheil, and B. Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering*, 2009.

[5] C. Atkinson, B. Kennel, and B. Goß. The level-agnostic modeling language. In B. Malloy, S. Staab, and M. Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin Heidelberg, 2011.

[6] C. Atkinson, D. Stoll, and C. Tunjic. Orthographic service modeling. In *EDOCW*, pages 67–70, 2011.

[7] M. Breu, R. Breu, and S. Löw. Moveing forward: Towards an architecture and processes for a living models infrastructure. In *International Journal On Advances in Life Sciences, IARIA*, volume 3, pages 12–22, 2011.

[8] J. de Lara and E. Guerra. Deep meta-modelling with metadepth. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Eclipse Foundation. OCLInEcore. `http://wiki.eclipse.org/MDT/OCLinEcore`, download June 2013.

[10] L. Engelen and M. van den Brand. Integrating textual and graphical modelling languages. *Electron. Notes Theor. Comput. Sci.*, 253(7):105–120, Sept. 2010.

[11] J. Espinazo-Pagán, M. Menárguez, and J. García-Molina. Metamodel syntactic sheets: An approach for defining textual concrete syntaxes. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 185–199, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] M. Eysholdt. Converging Textual and Graphical Editors. Eclipse Modeling Days 2009, 2009.

[13] Gentleware. Poseidon For DSLs. `http://www.gentleware.com/poseidon-for-dsls.html`, download April 2013.

[14] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, 1 edition, 2009.

[15] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 114–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Itemis. XText. `http://www.xtext.org`, download April 2013.

[17] Jet Brains. Jet Brains MPS. `http://www.jetbrains.com/mps/`, download April 2013.

[18] L. C. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, 2010.

[19] M. Lankhorst, H. Proper, and H. Jonkers. The architecture of the archimate language. In T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, Lecture Notes in Business Information Processing, pages 367–380. Springer Berlin Heidelberg, 2009.

[20] J. Lara and E. Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos, editors, *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 259–274. Springer Berlin Heidelberg, 2012.

[21] J. Lara and H. Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2002.

[22] A. Ledeczi, M. Maroti, A. Bakay, and G. Karsai. The generic modeling environment. In *WISP'2001*, 2001.

[23] Microsoft. Visual Studio Visualization and Modeling SDK. `http://archive.msdn.microsoft.com/vsvmsdk`, download April 2013.

[24] Object Management Group (OMG). Diagram Definition Version 1.0. `http://www.omg.org/spec/DD/`, 2012.

[25] F. Pérez Andrés, J. Lara, and E. Guerra. Domain specific languages with graphical and textual views. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 82–97. Springer-Verlag, Berlin, Heidelberg, 2008.

[26] A.-W. W. Scheer. *Aris-Business Process Frameworks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1998.

[27] M. Scheidgen. Textual modelling embedded into graphical modelling. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture âĂŞ Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 153–168. Springer Berlin Heidelberg, 2008.

[28] C. Schneider. On Integrating Graphical and Textual Modeling. Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel, 2011.

[29] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[30] J.-P. Tolvanen and M. Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 92–93. ACM, 2003.