

# Building Tools by Model Transformations in Eclipse

Audris Kalnins, Oskars Vilitis<sup>1</sup>, Edgars Celms<sup>1</sup>, Elina Kalnina<sup>1</sup>, Agris Sostaks<sup>1</sup>, Janis Barzdins  
University of Latvia, IMCS, 29 Raina boulevard, Riga, LV-1459, Latvia, +371 7224 363

Audris.Kalnins@mii.lu.lv, Oskars.Vilitis@gmail.com, Edgars.Celms@mii.lu.lv,  
Elina.Kalnina@lumii.lv, Agris.Shostaks@gmail.com, Janis.Barzdins@mii.lu.lv

**Abstract.** The paper is devoted to metamodel-based graphical tool building platforms, the main applicability area of which would be the development of DSL editors. In the beginning, a brief analysis of the well-known static-mapping-based approach is given and situations when it is insufficient are pointed out. Then, in the main part, a new, more flexible approach based on model transformations is proposed, where the correspondence between the domain metamodel and the presentation metamodel is defined by transformations in an appropriate transformation language (MOLA). The paper presents a newly developed Eclipse plugin METAclipse, which defines the presentation metamodel consisting of several parts, the most important of which is the graph diagram metamodel. METAclipse provides engines driven by these metamodels, which are incorporated in a very simple architecture allowing easy use of transformations. METAclipse has proven its flexibility and efficiency in the development of a new generation graphical editor for MOLA itself.

**Keywords:** DSL editors, model transformations, metamodel-based graphical tool, Eclipse

## 1 Introduction

Due to the advent of the MDA approach and the growing popularity of various domain-specific languages (DSLs), various graphical tool building environments have gained continuously increasing attention in recent years. The first simple generic metamodel-based tool environments, such as MetaEdit [1], Kogge [2] and early versions of Dome [3] and GME [34], appeared already in the mid-nineties, but their capabilities were quite limited.

The second generation of such metamodel-based environments with much wider possibilities, such as MetaEdit+ [4], GME [5], and ATOM3 [6], appeared around 2000 (the first version of MetaEdit+ actually appeared much earlier [35]). They already had domain metamodeling facilities close to MOF [7] and more advanced graphical capabilities. Therefore the popular tool paradigm of a visual language being based on a presentation-independent domain (as it is e.g., for UML[16]) could be supported. But the presentation metamodel (the description of graphical elements) still had to be close to the domain metamodel, only relatively simple mappings between them were permitted, and everything else had to be defined by OOPL code (e.g., C++ in GME). The previous tool framework by UL IMCS, the Generic Modeling Tool environment [8], also belongs to this category.

A completely new generation of tool frameworks has emerged in recent years as a response to the need by MDA community to make DSLs an everyday software development practice. One such group of environments is based on the open-source Eclipse platform. Eclipse, together with its EMF plugin [9], is a broadly used metamodeling environment, close to MOF. In addition, the GEF plugin [10] is a basic “diagram drawing engine.” Only something linking the two was required for a complete tool building environment. The first and most popular solution for this is the static metamodel mapping-based GMF platform [11], which is analyzed more closely in section 3. Alternative solutions are provided by Pounamu/Marama [23] environment and the coming GEMS project [30].

A popular alternative to Eclipse on a commercial basis is offered by Microsoft DSL Tools [24] in Visual Studio 2005, however the logical capabilities there are quite close to GMF. The already mentioned MetaEdit+ [4] has significantly evolved and also become a key player in this area.

---

<sup>1</sup> supported partially by ESF (European Social Fund), project 2004/0001/VPD1/ESF/PIAA/04/NP/3.2.3.1/0001/0001/0063

The abovementioned solutions are quite appropriate for simple cases, where the domain and presentation metamodels are close and no complicated mapping logic is required. However, DSL support frequently requires much more complicated and flexible mapping logic. Therefore a new approach has appeared: to define this mapping by *model transformation languages*. Model mappings in tools actually lie very close to traditional MDA tasks, for which model transformation languages were invented. Therefore they can be treated as very appropriate DSLs for metamodel-based tool building, yielding development efficiency that is an order of magnitude higher when compared to that of OOPL.

The first frameworks using this approach to a degree are the Tiger project [12] and the ViatraDSM framework [31]. Both are based on Eclipse and use GEF as a drawing engine. The Tiger project is based on the graph transformation language AGG [13]. However, a specific domain modeling notation is used there which forces the domain metamodel of a language still to be close to the presentation metamodel. Standard editing actions (create, delete, etc.) are specified by graph transformations which act on the domain model, and the presentation model is updated accordingly. The ViatraDSM framework is based on the Viatra2 transformation language [32]. There also the domain metamodel must be close to the presentation one, but a larger freedom is allowed, and the transformation approach to a degree can be combined with the static mapping approach. There are also plans to use Fujaba [33] transformation language in the MOFLON framework [36].

The next section provides a more detailed discussion on the two approaches to tool frameworks and their respective merits.

In this paper the *METAclipse* plugin for Eclipse is proposed, which is a *completely transformation based* tool framework. There are no restrictions on the correspondence between the domain and presentation metamodels. The mappings are defined dynamically by *transformations* in the model transformation language MOLA [14]. The implementation of METAclipse reuses the basic Eclipse components such as EMF and GEF, as well as parts of GMF runtime [25]. The main distinguishing feature of METAclipse is an appropriately built presentation metamodel, which is discussed in some detail in section 5. It enables a clear separation of responsibilities between the METAclipse presentation engine, which handles all the low-level presentation and layout-related tasks (all GEF-level concepts are hidden), and transformations, which create and maintain only the domain and the logical structure of presentation. METAclipse contains also a universal metamodel-based presentation engine for element property editing, and an advanced project tree engine. The logical behavior of property dialogs and model tree also is controlled by appropriate transformations, thus transformations control the complete tool behavior. The METAclipse engines are discussed in section 6. The usability of the approach is ensured by the fact that a significant part of the transformations are domain-independent and are built only once, as part of the framework itself. This possibility is ensured by the *scaffolding* part of the metamodel (see details in section 7). The paper concludes with some experience of using METAclipse.

## 2 Visual DSLs and two paradigms for tool building frameworks

### 2.1 General principles of paradigms

A visual language basically consists of two parts – the domain part and the presentation (visual) part. Sometimes they are called also the abstract and concrete syntax respectively - the terminology taken from textual languages. For visual languages the mostly used syntax specification technique is metamodels. Sometimes graph grammars are used too (this possibility will not be analyzed here).

The domain part of the language is defined by means of the *domain metamodel*, where the relevant language concepts and their relationships are formalized. The domain metamodel is used also for precise definition of language semantics. Typically standard MOF [7] is used for the definition of domain metamodel. Some frameworks [4,12,24,31] use a slightly alternative notation for domain metamodels, where those metamodel classes, which correspond to edges in diagrams, are singled

out and called relationships (actually they are equivalent to UML association classes, which formally are not part of MOF). A note on terminology should be added here. Though according to general modeling principles the language domain is defined by its metamodel, several frameworks, including GMF and MS DSL, call the definition the *domain model*.

For the presentation part (concrete syntax) there is no universally accepted notation. The same metamodeling techniques typically is used, but with various semantics. During the definition of the presentation part both GMF and MS DSL use a *presentation metamodel*. Instances of classes in this metamodel are *types* of diagram elements to be used in the diagram (e.g., ClassNode, AssociationEdge). A concrete set of graphical element types for a diagram definition is called the *presentation* (or *graphics*) *model*. During runtime GMF uses another metamodel (*notation metamodel*), where instances are specific graphic *elements* in a diagram (nodes, edges, etc.).

Most of the abovementioned frameworks, including GMF and MS DSL, use a *static-mapping*-based approach for definition the desired correspondence between the domain and presentation models. During the tool design this mapping assigns a presentation model element (a node type, edge type or label type) to a domain model element, by means of which the latter one must be visualized. Typically, a given node type is assigned to a domain model class and a given edge type to an association (or relationship). This means that all instances of the class will be visualized by the given node type. Just because of this fact it is fair to call the mapping static. The element structuring (what is within what, what can be connected to what, etc.) is also specified by these mappings. Thus, the complete graphical tool functionality is basically defined by this mapping (certainly, there is more - property dialogs, model browser etc, but we omit all this for a moment). The mapping may be complemented by some use of constraints (OCL [15] in GMF, custom languages in MS DSL and MetaEdit+), and this offers a limited flexibility to mappings. However, for example, you cannot visualize the same domain class sometimes as a node and sometimes as an edge.

Most of the frameworks (GMF, MS DSL, ) use the *generation step*, by means of which language classes are generated in the corresponding OOP language (Java, C#,...) from the involved models. The generated language classes implement the required functionality (certainly, also by inheriting from various "generic" classes). If the generated functionality is insufficient, the language code can be extended manually. Actually, static mapping may be used without the generation step too - examples are MetaEdit+ [26] and Generic Modeling Tool, which are model interpreters.

An alternative to the static mapping approach is the *model transformation* based approach, which is the main topic of this paper. According to this approach the correspondence between the domain and presentation models is defined dynamically by means of model transformations in an appropriate *model transformation language*. These transformations define what modifications must be done in one of the models, if the other changes (due to user actions or other internal activities). The synchronization may go both ways. The main difference to the static mapping approach is that transformations may implement any kind of data-based correspondence between domain and presentation instances. It should be noted that the transformation approach requires a different model-metamodel terminology (transformations always process models defined by metamodels). Therefore in the transformation approach by domain model we must understand a concrete set of runtime instances, and by presentation model a diagram (or several diagrams) representing this set visually. The definitions always are metamodels.

We conclude the section with some more comments on metamodeling. The defined domain metamodel always is at the layer M2 according to the MOF hierarchy (it is defined according to MOF itself, which is at M3). A concrete domain model (a runtime model in a tool) is at M1. However, in the static mapping approach the presentation definition metamodel is at M2 within the MOF hierarchy (it is not MOF itself, but a specific metamodel); the defined presentation model (which contains the diagram element types) is at M1. The runtime presentation model (also at M1) is not an instance of this definition model. This MOF layer mismatch is not a problem for the generation approach. There only one metamodel/model layer is used (metamodels for definition, corresponding models for generation of language classes). However, the transformation approach (for all transformation languages) requires a symmetric situation at both domain and presentation. Therefore the presenta-

tion definition metamodels from GMF, MS DSL or other frameworks cannot directly be used for transformations.

## 2.2 Areas of applicability

Here a brief discussion on DSLs is presented, in order to find out in what situations which of the approaches is superior.

Most of "engineering" DSLs (for telecommunications, process control, automotive industry etc) are quite simple from the point of view we are discussing now. There are typically no metamodeling standards in these domains, and domain metamodels can be simply extracted from graphical notations existing in practice. These are the typical DSL examples used in framework tutorials (see, e.g. [37]). There the mapping is very straightforward - domain classes map to nodes, associations to edges, class attributes to labels and so on. Certainly, their execution semantics is complicated, but this is another dimension. Static mapping based frameworks are an appropriate solution for this area. However, it should be noted at once that mapping based solutions are so easy only in the case when there is such a straightforward correspondence between the domain and presentation models. If some code is to be added manually, a deep knowledge of the generated code and various APIs of the framework itself are required, especially it is true in the case of GMF.

DSLs in software engineering area typically are more complicated in the aspect we are interested in. This is because of several reasons. Frequently there are domain metamodel standards for the given language, which are used for the formal semantics definition and are very far from the concepts directly used for the graphical presentation of the language. In addition, the interdependency between language elements is significantly more complicated, including dependencies between elements visualized in different diagram types. A typical desire for language tools there is that editing should be syntax directed - only valid language diagrams should be allowed, as far as possible. All these factors make the mapping between domain and presentation elements quite complicated, frequently many-to-many and data dependent. For example, a value of a domain class attribute may determine what other domain classes should also contribute to the same presentation element. All this makes the static mapping approach significantly less efficient, while all these situations pose no problems for the transformation approach. Let us give some examples.

The first such language class could be model transformation languages, more precisely, the graphical ones. They include the OMG standard MOF QVT [21], the MOLA language [14] used in this paper, Fujaba language [33] and some others.

Let us analyze a small excerpt from the MOF QVT Relations Language [21], for which the OMG standard proposes a domain metamodel, and both graphical and textual concrete syntax. The proposed domain metamodel is indeed quite far from the graphical form of the language. For example, an instance of the domain metamodel class `ObjectTemplateExp` (see [21], page 30) must be mapped to a pattern element (a rectangle, something like a class instance in UML). This domain instance may contain several instances of `PropertyTemplateItem`. Each contained instance must be mapped either to a slot in the element (a label within a compartment in GMF terminology) when the `value` link in the domain points to a proper OCL expression, or to a link (edge) to another pattern element (when the `value` link points to another `ObjectTemplateExp`). Another complicated issue here is building of label texts, which are composed from several domain classes and the composition is data-dependent.

It should be clear that the described correspondence between domain and graphical presentation cannot be described by a static mapping of any kind. This is not the only example of this kind in MOF QVT. If we tried to define this correspondence in GMF, this would mean vast programming in pure Java (with just some generated classes to be reused). On the other hand, by means of model transformations such a correspondence may be described quite easily. It should be noted, that currently there is no proper graphical tool for the relational MOF QVT.

The situation with MOLA language [14] (the basic language used in our approach) is similar, though the domain metamodel there is closer to the graphical presentation. However, the require-

ment that the graphical tool must be syntax directed (and support a lot of context-sensitive syntax constraints during diagram building) requires that the domain metamodel must be "semantic" to a great degree. Therefore the correspondence between domain and presentation is complicated in MOLA too. For example, each pattern element in MOLA domain (a concept similar to that in MOF QVT) has only one of several possible paths in a model to the corresponding class, the existing path must be found to display the class name as a label in the corresponding node. The composition of label texts is as complicated as in MOF QVT. A specific issue in MOLA tool (more related to property editing than mapping to presentation) is the requirement to offer only relevant associations for building a pattern link. To find the relevant list, a transitive closure of inherited associations in a class diagram must be computed. There are other such complicated situations for property editors in MOLA too. All the mentioned would make the implementation of MOLA tool by means of static mappings quite complicated. At the same time, this paper shows in section 8 that a MOLA tool satisfying all requirements has been built by reasonable efforts using the transformation approach.

One more example from a completely different area would be a tool which would visualize an RDF [19] data base as a simple class diagram, with update support too. This task has become popular in the context of ontology development. The domain model implied by typical RDF data bases [29] is a simplified metamodel for RDF triples [20]. To visualize RDF data as a class diagram (containing also instances) in a natural way, a complicated analysis of attribute values (of String type) for some RDF domain classes must be done. Only this way it may be decided whether an RDF property should be visualized as an attribute or association. Such a tool is apparently easier to be implemented by the transformation approach.

### 3 Reuse of GMF Elements in METAclipse

As it was already noted, GMF (Graphical Modeling Framework) is the most popular metamodel-based graphical tool building platform for Eclipse. GMF utilizes Eclipse EMF (Eclipse Modeling Framework) and GEF (Graphical Editing Framework) technologies. EMF is used for model management and GEF for graphical user interface.

GMF uses a *static-mapping*-based approach. It defines a set of metamodels: graphical (presentation), tooling and mapping metamodels. In addition, it uses ECore as the domain metamodel (in GMF terminology, it is meta-metamodel according to MOF). Graphical metamodel defines the graphical element types. Tooling metamodel defines the palette and menus. The mapping metamodel defines the mapping possibilities between the models. This mapping has restrictions - for example, nodes can be mapped only to EClasses (i.e., domain classes) and links to EStructuralFeatures (in fact, associations).

To build an editor in GMF, domain, graphical, tooling and mapping models are defined, then the generation is performed and manual code in Java added.

As it was already noted at the end of section 2.1, the graphical (presentation) metamodel is well adapted to the generation step in GMF, but cannot be used directly by the transformation approach. The same situation is with the tooling metamodel. Therefore actually nothing of the GMF definition part can be reused in the proposed METAclipse approach. As a consequence, there are no explicit graphical element types to be used by transformations (see section 5 for the METAclipse solution).

Fortunately, the GMF runtime [25] uses another metamodel - the notation metamodel. This metamodel describes graphical instances in the runtime - nodes, edges, compartments and labels (exactly, the layer required by transformations to build graphical objects dynamically). In fact, the GMF runtime is a graphical engine for Eclipse, significantly extending GEF in the direction required for diagram building.

METAclipse graph-diagram engine partially reuses components of the GMF runtime. The presentation metamodel in METAclipse is also partially based on GMF notation metamodel, only the metamodel elements are made more "transformation-ready". DSL tools also require rich possibilities for displaying domain elements in the project tree. In addition, editing of properties of domain

elements frequently is a semantically complicated operation, especially, the building of the most adequate lists to be prompted. GMF does not provide much support in this area. METAClipse, on the other hand, adds an extensive control over these editor parts and provides original components addressing these issues (see section 6 for more details).

## 4 Model-Transformation-Based Approach

In this paper we propose a completely transformation based approach to tool building, implemented in the METAClipse framework. The correspondence between domain and presentation models there is defined dynamically by transformations in a model-transformation language.

At the instance level this correspondence must be stored in the model - by means of *mapping links*, and these links are built and used by the transformations. Our experience shows that transformation building is significantly eased by the existence of such links.

Let us assume that a domain is given for which we need to build a tool, e.g., the Class domain in UML2. In METAClipse the domain metamodel is specified as an EMOF-compliant (Essential MOF, see [7]) UML model (similarly to the ECore model in GMF). The presentation metamodel is predefined in METAClipse (an equivalent to the notation metamodel in GMF). It describes the available graphical elements in a diagram: nodes, edges, text labels, and other elements. More precisely, an instance of this metamodel is a runtime graphical model, which is visualized and serviced by the METAClipse presentation engine. Transformations build instances of the presentation classes and connect them by mapping links to the corresponding domain instances. The presentation engine visualizes these presentation instances and, in addition, notifies transformations when the user has performed some action. A high-level graphical model transformation language MOLA [14] is used in METAClipse for building the required transformations, but in principle other such transformation languages could be used too, e.g., MOF QVT. There is no generation step in METAClipse (certainly, MOLA transformations must be compiled to an executable form).

Fig. 1 shows the most interesting part of our presentation metamodel. In fact, the complete metamodel of the world transformations have to work in consists of the predefined presentation metamodel (of which a small fragment is given in Fig. 1) and the required domain metamodel (of which only the `Element` class is visible). For UML-related domains (such as the above mentioned Class domain) and many other it is typical that there is a common ancestor for all domain classes (in UML it is `Element` in the `Kernel` package). The association `domain-present` here serves as a generic mapping link between domain and presentation.

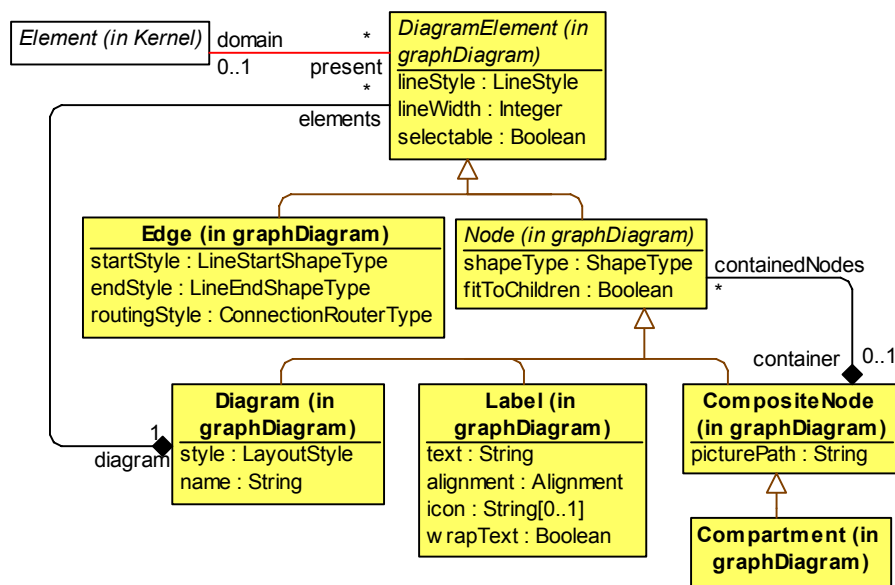


Fig. 1. Graph diagram metamodel

To implement the abovementioned schema, a transformation procedure must be built for each kind of diagram element (node, edge, or sub-diagram) representing an essential domain element. For example, such procedure has to be created for a class node (representing a domain `Class`). In particular, this transformation builds a new class node (an instance of `CompositeNode`) when a `Class` instance in the domain is to be shown in a class diagram.

Though building the transformation procedure for each diagram element may initially seem to require much more effort than just defining a static mapping in GMF, the job is actually quite easy in an appropriate transformation language. It should be reminded here that for DSLs we are interested in nearly each static mapping in GMF would need to be complemented by a relevant piece of Java code after the generation step. The transformation procedure can also collect in a cohesive block the various constraints relevant to a diagram element. These constraints become much more readable in MOLA than they would be in a mix of OCL and Java. The main gain, however, is flexibility.

Besides the domain-determined specific functionality for each diagram element, a lot of functionality in a tool (setting an element style, displaying menu items, moving an element etc.) is actually common to all elements, and therefore needs to be built only once, as part of the framework. The presentation engines in METAClipse perform all generic presentation and graphics-related jobs, such as moving an element. In order to maximize this common part in transformations, the presentation metamodel must be used in an appropriate way, especially the “scaffolding part” of it.

The next sections present the basic details of METAClipse implementation.

## 5 Structure of METAClipse Presentation Metamodel

As it was already pointed out, the METAClipse framework contains a domain-independent presentation metamodel. This metamodel serves as a *contract* between the presentation engines and the transformations, clearly separating the duties of each. The presentation metamodel in METAClipse consists of several logical parts, defined as packages. The `graphDiagram` package, whose most essential part was shown in Fig. 1, defines the available diagram element kinds in METAClipse. The *diagram presentation engine*, which will be described in some detail in the next section, visualizes any valid instance of that metamodel.

Another part of the metamodel, also closely related to presentation engines, is the commands part, shown in Fig. 2. The `Command` class represents a request generated by a presentation engine to be serviced by a transformation. A *command* is a direct result of some tool user action, which is not “purely visual” (layout-related) and therefore requires a transformation for its execution. The context in which the command is to be executed is specified by appropriate links. The metamodel in Fig. 2 shows the basic commands related to graph diagrams. *Create* commands are generated by the engine when the user selects a palette element and clicks on the diagram. It is up to transformations to decide whether the user request is semantically valid. It should be noted that the command concept in METAClipse is at a higher level of abstraction than a similar concept in GMF and therefore easier for implementation by transformations.

Another command in the fragment is *select*, in response to which a transformation must build the appropriate property dialog (to be visualized by property presentation engine).

The palette and menu parts of the metamodel (and actually also a certain fraction of the property dialog) have the special feature that their instances *do not change* during the tool action: they are in fact part of a specific tool definition.

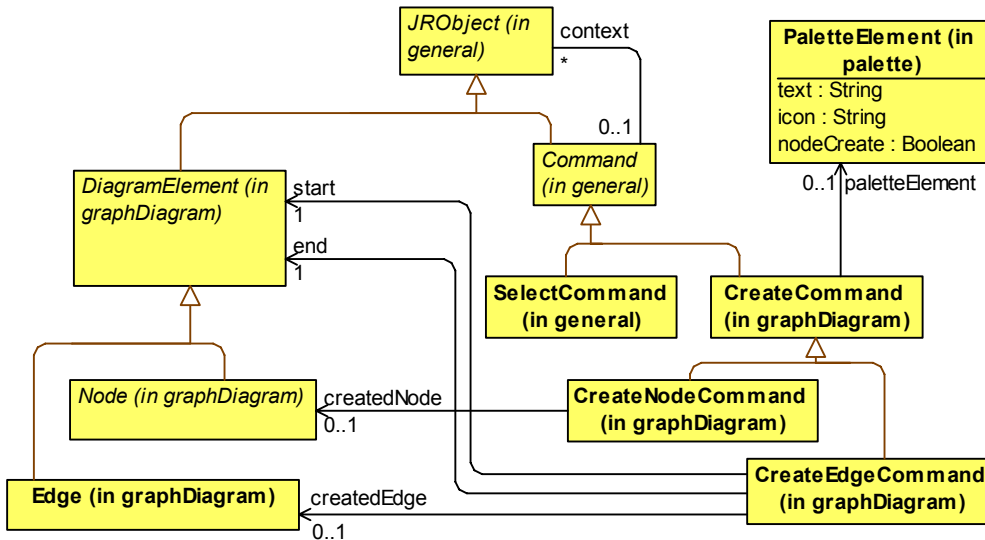


Fig. 2. Command part of the presentation metamodel

In order to simplify the transformation building in METAclipse and imitate the “static mapping paradigm” of GMF as far as possible, there is another static part in the predefined metamodel, which is used by the transformations only: the *scaffolding* part. This part is used to define the structure of a specific diagram (e.g., a UML class diagram) in a static way, and to “hook up” by links on this “static frame” all static parts of the metamodel, such as palette and menus. Fig. 3 shows that “static” part of the metamodel and some of its links to the other (“non-static”) component. The scaffolding consists of two subparts. One is the diagram structure definition by means of instances of DiagramType, NodeType, EdgeType etc., which constrain what types may be inside/under other types in a valid diagram. This structure definition is a semantic equivalent to the static presentation definition in GMF, but in a form transformations can benefit from. The other part is style definitions, which specify the graphical styles for diagram elements.

Instances of the static part are built only once, at the start of a new project. The role of scaffoldings for building transformations will be conveyed in some detail in section 7.

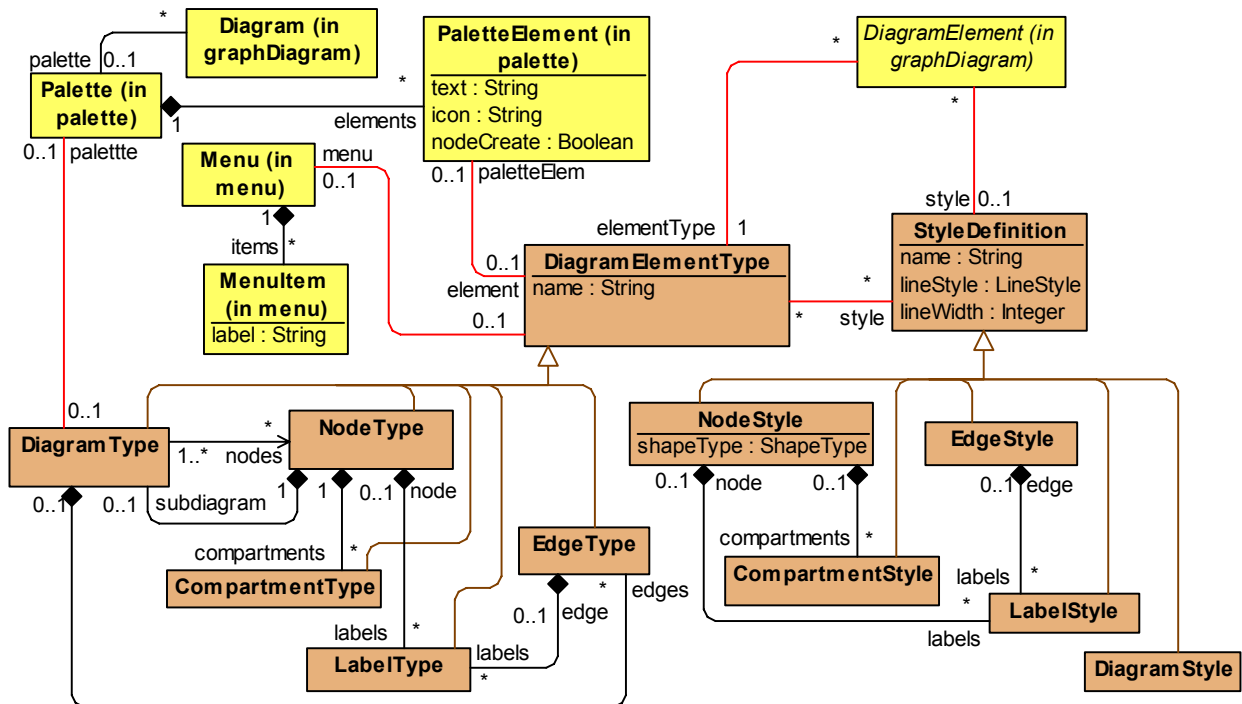


Fig. 3. Static part of the metamodel



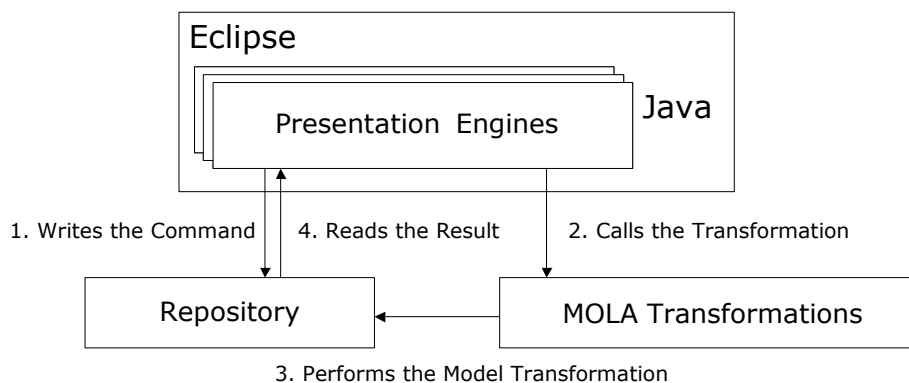
When a tool is to be defined by means of the METAClipse framework, a concrete domain meta-model is to be added to the predefined (presentation) metamodel and some mapping associations must link the domain to the presentation part (the most important is the one depicted in Fig. 1). Then the domain specific MOLA transformation procedures can be added to the fixed set of the transformations that are part of the METAClipse itself.

## 6 METAClipse Presentation Engines

The overall architecture of METAClipse is very simple. METAClipse consists of several parts, each of which represents a separate engine and “understands” part of the metamodel discussed in the previous section (“knows” how to interpret the models of this metamodel). The processing loop of every event in each METAClipse presentation engine can be seen in Fig. 4. It also depicts the main parts of METAClipse; besides presentation engines they are MOLA transformations and model repository. The repository is the central model storage and is required for transformations to run. Transformations are compiled procedures of the MOLA transformation language, which uses repository API to perform model changes.

First, after any event has been triggered (new node added to diagram, context menu called, selection changed, etc.), the corresponding command is written to the repository, describing the logical event that has happened. The command represents a semantic event, one of these defined in the presentation metamodel (see section 5).

Next, the transformation is called, which then performs the necessary model transformation for the command written in repository. Transformation can change both domain and presentation models. After the execution of the transformation, engines examine the model changes and the changed model elements are once again read from the repository. All changes to visual model elements lead to changes in user interface.



**Fig. 4.** High-level view of the METAClipse architecture

This simple event processing loop is used consistently over all presentation engines and this HTTP-get-like request-response protocol defines the interaction between presentation engines and transformations.

For example, when a user wants to create a new class and chooses the relevant palette element, a new `CreateNodeCommand` instance is created in the repository. The transformation now creates the relevant domain and presentation instances, and the diagram engine visualizes the new `CompositeNode` instance.

As outlined above, METAClipse consists of several presentation engines. The main and noteworthy engines are the following:

- Graph diagram engine – the main part of the METAClipse presentation engine, based on GEF and some components of GMF and providing runtime for diagramming.

- Property engine – the extension of Tabbed Properties Eclipse view. Defines a consistent metamodel and provides means for editing domain element properties.
- Project explorer engine – the extension of Common Navigator Eclipse view. Also defines a consistent metamodel and provides means for representing elements in the project tree.

An original idea has been used in METAClipse for accessing instances in the transformation repository from EMF. This access ("Wise Objects") has been implemented in a universal way, as a special extension of EMF object. In the result all EMF features and services can be freely used in METAClipse. The implementation includes also an efficient change notification. Only due to this access mechanism, both transformations and engines can jointly manage presentation instances.

## 7 Transformation Structure

As it was already mentioned, the transformation set in METAClipse consists of two parts: the domain-independent part, which actually belongs to the framework itself, and the domain-dependent part, which implements functionality specific to the graphical editor for some domain and should be built for each new domain. As it was explained in the previous section, transformations in totality have to execute the current command (only one at a time) generated by some presentation engine.

The domain-independent part includes an interpreter-like structure, which recognizes the current command and checks its overall validity. But the main volume of domain independent procedures is based on the static part.

The static part of the metamodel (Fig. 3) is created by special initialization transformations, which are invoked once per project, immediately after its physical creation.

When instances of the static part are in place, many transformation tasks can be implemented in a domain-independent way, as generic interpreters based on static instances. The first such example is style-setting transformations, which set default styles for new diagram elements or perform data-dependent style modifications when domain instance data have been modified (such as the appropriate style for an association end). Certainly, all possible style instances must be built during the initialization. A similar schema can be used, for example, to build a universal procedure for displaying menu items for the selected diagram element.

The domain-dependent transformation procedures do domain-dependent semantic jobs (build domain and presentation instances, build details of a property dialog, update object properties and so on). These actions cannot be performed by universal domain-independent procedures, since a lot of specific semantic checks are to be performed during such editing. Nevertheless, these specific procedures can use a lot of domain-independent subroutines for standard jobs, such as a universal "context validity" checker for a new element (on the basis of the static structure definition), a presentation element remover (used after a domain element was deleted, which may involve nontrivial semantic checks), and others.

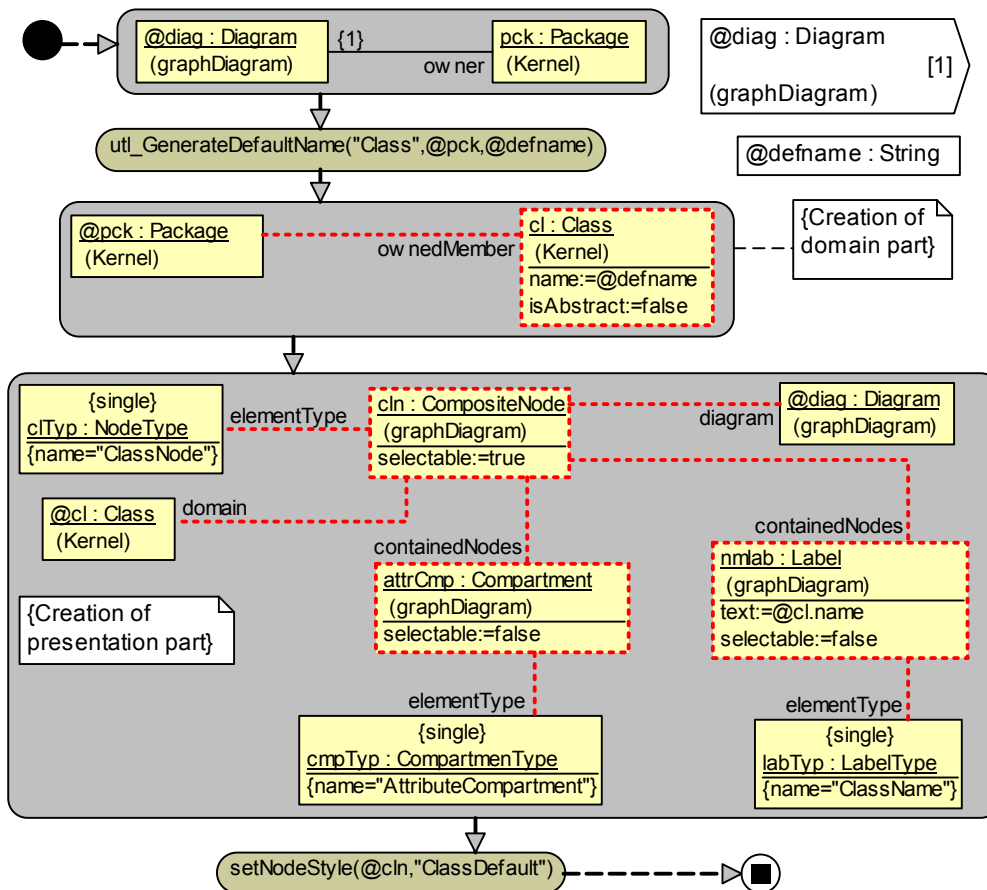


Fig. 5. MOLA transformation example - create a new class

To assess the size of transformations, we conclude the section with a MOLA transformation example (Fig. 5, see [14] for MOLA syntax). This is the transformation that creates a new class and a class node for it. Just to have some impression, what is going on there - grey rounded rectangles are rules, rectangles with solid borders denote pattern elements to be matched and those with dotted borders denote new instance creation, lines correspond to metamodel associations.

## 8 METAcclipse Experience

The first version of METAcclipse has been implemented at UL IMCS as an Eclipse plugin, using MOLA as the transformation language. The main testbed for this version has been a new editor of MOLA itself, since MOLA is clearly in the DSL category where transformation based approach is better. The MOLA environment has been developed in a bootstrapping manner [18], with the previous prototype editor built using the Generic Modeling Tool [8] framework. The new editor implements a lot of validity checks and smart prompting during the diagram building. The UML class editor example discussed above actually is the simplest part of the MOLA environment, the metamodel editor. The MOLA procedure editor requires much more sophisticated domain-specific logic during element building or updates. Both editors are interdependent: for example, the modification of a class name must be reflected in all class element instances in MOLA rules that reference the given class. After the first version of METAcclipse was completed (including about 180 domain-independent MOLA procedures), the implementation of the new MOLA editor required about one man-month to develop and test (containing about 120 procedures in the domain-dependent part; there are about 30 essential classes in the domain metamodel). Thus the experiment can be considered a complete success.

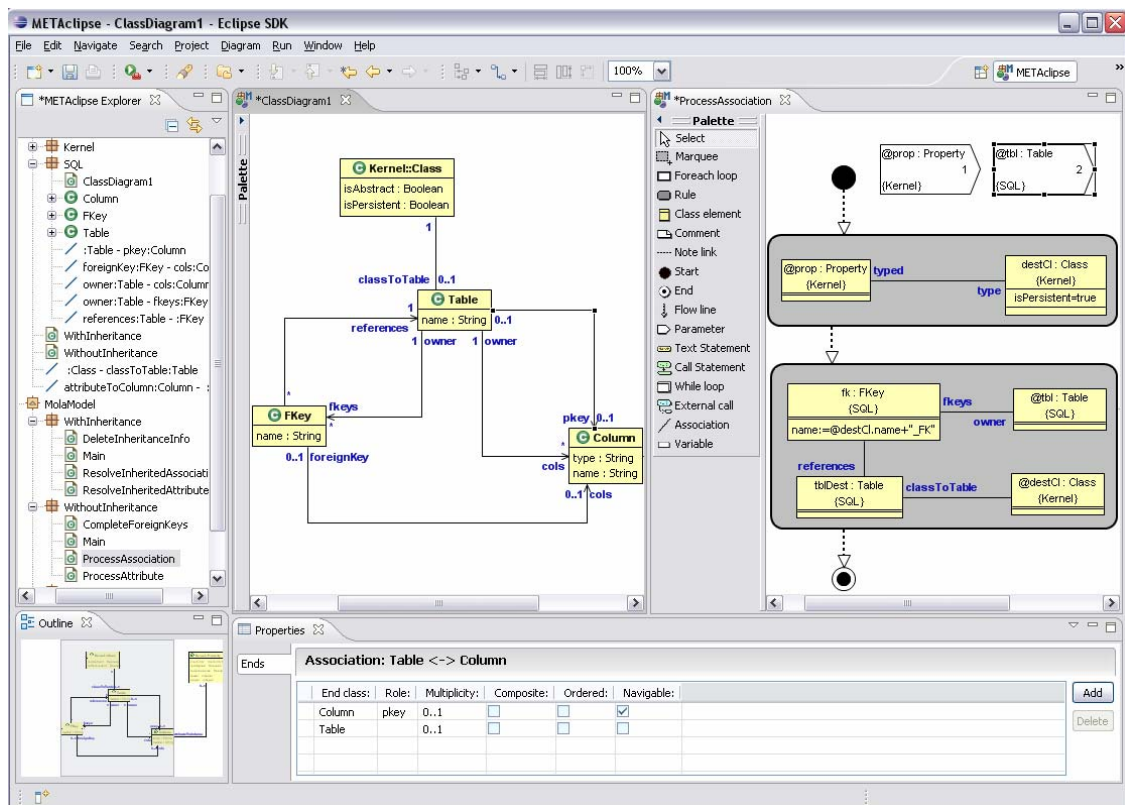


Fig. 6. MOLA editor implementation in METAcclipse

Fig. 6 shows the editor in action – with both a sample class and MOLA diagrams visible. The tool performance is completely satisfactory, it has been tested on examples of realistic size (about 80 MOLA procedures and 4000 domain instances total). In addition to editors, the MOLA tool contains also MOLA compiler (built in a lower level transformation language L3 [27], also developed at UL IMCS), which runs on the same repository. The developed MOLA tool is being successfully applied as the main "MDA instrument" in the European IST project ReDSeeDS [28].

The developed MOLA editor with its sophisticated semantic checks can be considered to be a typical complicated DSL example, and authors hope that this experience would be valid for other such DSLs too, for example a graphical MOF QVT tool.

## 9 Conclusions and Future Work

The paper describes the METAcclipse plugin, which provides a complete implementation of the transformation based approach to DSL tool building. The basic principles and metamodel and transformation structure is described. The paper has demonstrated that a model-transformation-based approach to tool building on the Eclipse platform is an efficient approach in cases when mapping between the domain metamodel and the presentation metamodel is complicated enough. The developed METAcclipse plugin has been tested on a non-trivial example – a new editor for the MOLA transformation language.

Certainly, any complicated DSL contains also simple parts where the static mapping based approach would be easier. Therefore both approaches should be combined. The difficulties there lie mainly in an appropriate building of presentation metamodel (taking into account the MOF layer mismatch mentioned at the end of section 2.1). Several ways to overcome these obstacles have been found - the inclusion of wizards which would generate automatically the required transformation procedures from the defined mappings, adding new "template-like" facilities or "meta-patterns" to MOLA transformation language and some other. The best way is still to be found, nevertheless, authors hope that this is doable.

## References

1. Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: Metaedit – a flexible graphical environment for methodology modelling. Springer-Verlag, 1991
2. Ebert, J., Suttentbach, R., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Spain, 1997, pp. 203-216
3. DOME Users Guide, <http://www.htc.honeywell.com/dome/support.htm>
4. MetaEdit+, <http://www.metacase.com/>
5. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason IV, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001
6. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM3. Software and System Modeling, 3(3), 2004, pp. 194–209
7. Meta-Object Facility (MOF), <http://www.omg.org/mof/>
8. Celms, E., Kalnins, A., Lace, L.: Diagram definition facilities based on metamodel mappings. Proceedings of the 18th International Conference, OOPSLA'2003, Workshop on Domain-Specific Modeling, Anaheim, California, USA, October 2003, pp. 23-32
9. Eclipse Modeling Framework (EMF, Eclipse Modeling subproject), <http://www.eclipse.org/emf/>
10. Graphical Editor Framework (GEF, Eclipse Tools subproject), <http://www.eclipse.org/gef/>
11. Graphical Modeling Framework (GMF, Eclipse Modeling subproject), <http://www.eclipse.org/gmf/>
12. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages using Tiger. Proceedings of GraBaTs'06, 2006, pp. 12
13. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), Vol. 3062, Springer LNCS, 2004
14. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. Proceedings of MDFAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62-76
15. OMG, Object Constraint Language, version 2.0, <http://www.omg.org/docs/formal/06-05-01.pdf>
16. OMG, Unified Modeling Language: Superstructure, version 2.0, <http://www.omg.org/docs/formal/05-07-04.pdf>
17. UML2 Tools (part of Eclipse Model Development Tools project), <http://www.eclipse.org/modeling/mdt/>
18. UL IMCS, MOLA pages, <http://mola.mii.lu.lv/>
19. Resource Description Framework (RDF), <http://www.w3.org/RDF/>
20. Ontology Definition Metamodel (ODM), Final Adopted Specification, OMG. <http://www.omg.org/docs/ptc/06-10-11.pdf>
21. MOF QVT Final Adopted Specification, OMG, document ptc/05-11-01, 2005.
22. Business Process Modeling Notation (BPMN), Final Adopted Specification, OMG. <http://www.omg.org/docs/dtch/06-02-01.pdf>
23. N. Zhu1, J. Grundy and J. Hosking. Pounamu: a meta-tool for multi-view visual language environment construction. Proc. IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC'04), pp. 254-256, 2004.
24. S. Cook, G. Jones, S. Kent and A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.
25. R. Gronback, Build Better Graphical Editors with the Graphical Modeling Framework, Slides, Eclipseworld 2006, [http://wiki.eclipse.org/images/0/08/Gronback\\_EclipseWorld2006\\_GMF.ppt.zip](http://wiki.eclipse.org/images/0/08/Gronback_EclipseWorld2006_GMF.ppt.zip)
26. MetaEdit+ Method Workbench User's Guide, Version 4.0, <http://www.metacase.com/support/40/manuals/mwb40sr2a4.pdf>, 2005.
27. Lx Transformation Language Set, <http://Lx.mii.lu.lv/>, 2007.
28. ReDSeeDS. Requirements Driven Software Development System. European FP6 IST project. <http://www.redseeds.eu/>, 2007.
29. Sesame, <http://www.openrdf.org>, 2007.
30. The Generic Eclipse Modeling System (GEMS), <http://www.eclipse.org/gmt/gems/>
31. I. Rath, D. Varro. Challenges for advanced domain-specific modeling frameworks. Proc. of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France.
32. Visual Automated Model Transformations (VIATRA2), GMT subproject, Budapest University of Technology and Economics. <http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html>
33. Fujaba. Universitat Paderborn, Institut fur Informatik. <http://www.wcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf>

34. Karsai G.: A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming, IEEE Computer Society Press, pp. 36-44, 1995.
35. Steven Kelly, Kalle Lyytinen, Matti Rossi: MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment Lecture Notes in Computer Science, Volume 1080, Proceedings of the 8th International Conference on Advances Information System Engineering, pp. 1-21, Springer-Verlag, 1996.
36. C. Amelunxen, A. Königs, T. Rötschke, A. Schürr: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. Model Driven Architecture - Foundations and Applications: Second European Conference, Lecture Notes in Computer Science, Vol. 4066, pp. 361—375, Springer 2006
37. MetaCase, The S60 Phone Example, Version 4.5,  
<http://www.metacase.com/support/45/manuals/S60%20Phone%20Example.pdf>