DSL tool development with transformations and static mappings

Elina Kalnina, Audris Kalnins

University of Latvia, IMCS, Raina bulvaris 29, LV-1459 Riga, Latvia
Elina.Kalnina@lumii.lv, Audris.Kalnins@lumii.lv

Abstract. A tool development framework for domain-specific languages combining mapping and transformation based approaches is proposed in this research project. The combination of both approaches permits to use advantages and eliminate disadvantages as far as possible. First results are described including draft architecture for the framework implementing proposed ideas. A sketch of mapping definition facilities is presented. Initial implementation proposals are described as well. A template based graphical generation language Template MOLA for implementation algorithm description is introduced.

1 Introduction

Currently it is very popular to create and use specialized modelling languages for a domain area. Theses languages are called *domain-specific languages* (DSL). They are developed for users specialized in the concrete area. By using domain-specific languages users can operate with familiar terms. There can be graphical or textual domain-specific languages. Only graphical languages will be considered here. Operational semantics of DSL is also out of scope of this research project. A visual domain-specific 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 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 the precise definition of language semantics. Standard MOF [1] or similar notations are used for the definition of domain metamodel.

For the presentation part (concrete syntax) definition there is no universally accepted notation. The same metamodelling techniques are used, but with various semantics. Most frequently, instances of classes in the presentation type metamodel are *types* of diagram elements to be used in the diagram. A concrete set of graphical element types for a diagram definition is called the presentation type model (a typical example is the graphical definition model in GMF [2]).

Tool development for graphical domain-specific languages is time consuming and expensive task. Due to the growing popularity of domain specific languages various

graphical tool building frameworks have been developed to improve the tool (editor) building process. Two different approaches are used in these environments. The first option is to use a mapping-based approach. During the tool design this mapping assigns a fixed presentation type model element (a node type, edge type or label type) to a domain metamodel element, by means of which the latter one must be visualized. This solution is quite appropriate for simple cases, where no complicated mapping logic is required. In this case tools for simple DSLs can be developed even during a presentation session. However, DSL support frequently requires much more complicated and flexible mapping logic. One of the reasons is that there is no fixed correspondence between the domain metamodel and presentation types. In this case the second approach is used: to define the correspondence by *model transformation languages*. Transformations define the synchronisation between the domain and presentation models and the tool behaviour in general.

Mapping based frameworks are MetaEdit [3], GMF framework [2], Microsoft DSL Tools [4], Generic Modeling Tool [5] and some other. A pure transformation based framework is METAclipse framework [6]. The other transformation based frameworks Tiger GMF project [7], ViatraDSM framework [8] and GrTP [9] provide also some elements of the mapping based approach.

There exist mapping based and transformation based tools, but usually some parts of the same DSL are suitable for mappings and some for transformations. It means none of solutions is optimal. Problem is that there is no good combined solution. In this paper the combined solution problem is addressed.

The only framework which already proposes some sort of combined solution involving both mapping definition and transformations is ViatraDSM framework [8]. However, a lot of principal issues such as a generic mapping metamodel, seamless integration of static mappings with transformations and user-friendly mapping definition facilities are not solved there. Therefore new ideas for a really satisfactory solution for combined approach to tool building framework are required.

The given paper briefly proposes a new complex solution how to combine transformations and static mappings for tool building. The paper concentrates on architectural solutions and required language facilities. In particular, a new mapping definition language ensuring close mapping and transformation integration is proposed. Some ideas how to implement this solution are also sketched briefly. The implementation is planned as a natural extension of METAclipse framework [6], thus providing advanced facilities for defining presentation type model, mappings and other parts of the tool definition. An interesting facility is introduced for the transformation generation from mappings. It is proposed to use the Template MOLA language to specify the generation algorithm. Template MOLA apparently is the first template based graphical generation language and therefore has a value of its own.

As there is no universally accepted terminology in the area, the second section of this paper begins with a terminology clarification. This section continues with the description of mapping and transformation based approaches as well as related work. In section 3 the original ideas how to combine transformations and mappings are presented. In this chapter the new mapping definition language is introduced. Facilities required to implement this language are described in section 4. Template MOLA is briefly described at the end of section 4.

2 State of the Art in DSL Tool Development

In this section the existing approaches for DSL tool development are briefly described.

2.1 Terminology Explanation

Let us begin with some terminology clarification. Currently different DSL development frameworks use completely inconsistent terminologies, even the terms model and metamodel are used differently depending on the context. For example, the mapping-based GMF [2] speaks only of two layers: model and metamodel, everything the tool builder creates is termed model. This paper combines both the transformations and static mapping context. To avoid misunderstanding, a consistent terminology and its relations to be used in this paper are defined in Fig. 1.

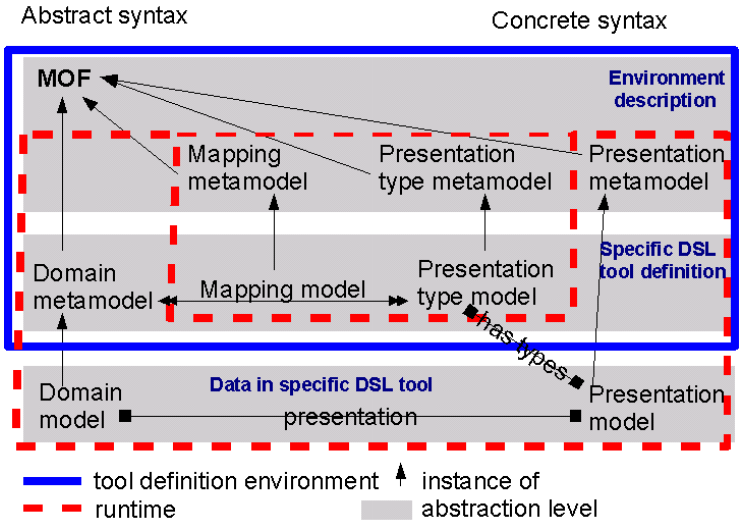

Fig. 1. Terminology definition

As we can see the domain metamodel is defined using MOF as a meta-metamodel. A domain model is created according to the domain metamodel. It should be noted that alternative domain meta-metamodels used in some approaches in fact play the same role as MOF (and are similar to it).

The situation is not so simple with the presentation part. In every framework there is a fixed presentation type definition environment. Possibilities supported in this environment can be described with a presentation type metamodel. Presentation types for a concrete domain specific language constitute a presentation type model defined according to the presentation type metamodel. Presentation types describe the relevant graphical element types. When data is created in this concrete DSL tool instances of presentation model are created, but data in this model is not an instance in the presentation type model. It is an instance of the presentation metamodel describing

supported graphical elements in the tool in general, for example, line, box, label etc. In the presentation type model, for example, we can describe that we want to represent this type as a grey rounded rectangle, with green lines and containing one label. In this case instances of the rounded rectangle, label and colours will be created in the presentation model, with appropriate properties set (according to the presentation metamodel). After instances have been created the user can change the rounded rectangle colour (if this feature is supported by the tool). In this case the presentation model is modified, but it does not affect the presentation type model. The presentation type describes only the default look of this node. That is why presentation model and presentation type model are two separate models.

One more important thing is to define a mapping model. This should be done according to the mapping metamodel. The mapping model describes the relationship between the domain metamodel and presentation types. At the data level mappings are not used directly.

When defining a new DSL tool in a tool definition framework, the user has to define a domain metamodel, a presentation type model and a mapping model. It should be noted that the presentation metamodel is needed directly only if mappings are defined using model transformations. Models required at runtime for the tool created from the definition depend on whether the tool definition framework is an interpreter or generator. If the framework is an interpreter the mapping and presentation type models are needed to interpret them in runtime. If the framework is a generator these models are not needed in runtime because the tool code is generated according to data in these models.

Most of the known DSL tool definition frameworks can be correctly categorized in the framework of this terminology schema.

2.2 Mapping-Based Approach

A *mapping*-based approach prescribes by means of which presentation type model element each domain metamodel element must be visualized. Thus, the graphical tool functionality is basically defined by this mapping. The mapping itself can be defined as a mapping model according to the mapping metamodel. The mapping typically may be complemented by use of constraints, but only at few selected points.

Most of the frameworks (GMF, MS DSL…) use the *generation step*, by means of which language classes are generated in the corresponding OOPL (Java, C#,…) from the involved models. The generated code ensures the relevant synchronization between the domain and presentation models in runtime. If the generated functionality is insufficient, the language code can be extended manually. Actually, mapping may be used without the generation step too - examples are MetaEdit+ [3] and Generic Modeling Tool [5], which are model interpreters.

It must be noted that the mapping approach is easy to use - if the generated code is sufficient (or should be accompanied by a small amount of manual code), the tool definition is mainly declarative and very fast. However, when the presentation type model is dissimilar to domain metamodel, a lot of code in an OOPL must be added. To avoid this, it is a common practice for simple DSLs to create custom domain metamodels nearly isomorphic to the corresponding presentation type metamodels

(one class to one node type and so on). However, there can be situations when it is not possible to select the domain metamodel freely, for example, if it is used for compiling, integration with other tools etc.

Mapping definition capabilities of a framework depend on mapping design patterns supported. The most expressive static mapping language is implemented in GMF. But even it is not expressive enough. For example, every domain class mapped to a diagram node must be contained in a domain class mapped to the diagram itself (canvas in GMF). Therefore it is impossible to implement by pure mappings standard UML class diagram where a class is contained in a package (in UML domain) and is visualised in several diagrams independently of its package containment.

Let us take a look at some DSL language examples where mapping approach is clearly insufficient. Evidently, one such group is model transformation languages. A typical example is MOLA [10, 11], which is a graphical language with a lot of semantic dependencies between language elements. It is important to use the native MOLA metamodel as a domain metamodel for the MOLA tool, since only this way complicated syntax checks can be performed during editing and context-sensitive lists of valid references proposed. If the goal of the tool is to create as syntactically correct models as possible, clearly it is impossible to implement this tool using only static mappings. The same can be said about tools for other transformation languages, for example, MOF QVT [12], where the native domain metamodel is even further from the presentation. Another such group could be complicated workflow languages.

2.3 Model Transformation Based Approach

A complete alternative to the mapping-based approach is the *model transformation* based approach. The correspondence between the domain and presentation is defined by *transformations* in a model transformation language, for example, MOLA [10, 11]. These transformations define what modifications must be done in one of the models, if the other one changes (due to user actions or other internal activities). Therefore the correspondence between the domain metamodel and presentation type model may be arbitrarily complicated here. In fact, transformations control the complete tool behaviour.

From the first glance this approach is more complicated to use - though experience shows that programming model element mappings in an adequate model transformation language is much easier than in a standard OOPL. The usability of the approach is ensured also by the fact that a significant part of the transformations are domain-independent and are built only once, as part of the framework itself. Clearly, the transformation driven approach is more time consuming in simple cases.

The first pure transformation based project is the Tiger project [7]. However, a specific domain modelling 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 main goal of Tiger approach is to provide the building of syntactically correct diagrams only.

The most advanced transformation based framework is METAclipse [6] which uses the MOLA transformation language and a powerful presentation engine in

Eclipse which is an extension of GEF, GMF runtime and some other plug-ins. It is based on a presentation metamodel specially adapted for defining transformations. The current version of MOLA editor [6] is built on this framework (using a bootstrapping approach). This editor provides an advanced support for ensuring syntactical correctness of MOLA programs and a high usability. The developed editor confirms the suitability of the framework for implementing complicated DSLs.

2.4 Combined Approach

Usually, for some parts of the tool the correspondence from domain to presentation is simple (fit for mappings) and for some complicated (fit for transformations). The best solution would be to combine both approaches. In this case for simple one-to-one relations between domain and presentation the mapping based approach could be used, but for complicated parts model transformations could be written. For example, for the abovementioned MOLA Editor [6] the transformation size could be reduced approximately by 50% if mappings were applicable. Simple visualisation could be defined by mappings, but for complicated consistency maintenance transformations would still be needed.

Currently there are only known two attempts to combine both approaches in a limited way. Frameworks using this combination to a degree are the Tiger GMF Transformation project [13] and the ViatraDSM framework [8].

The Tiger GMF Transformation project [13] (related to the original Tiger project) proposes to extend GMF by complex editing commands. The mapping between domain and presentation models is defined by standard GMF facilities. But new complex model editing commands can be defined by transformations acting only on the domain model. However, this approach does not permit to define more complicated (transformation based) mappings between the domain and presentation, which is the main goal of the approach proposed in this paper.

The ViatraDSM framework [8] is based on the Viatra2 transformation language [14]. In this framework a mapping from domain to GEF-level presentation concepts has to be defined. This static mapping is interpreted by the ViatraDSM engine. The transformation based mapping (defined by Viatra2 rules) can be combined with the static mapping approach. The goal of ViatraDSM seems to be the closest to the proposal in this paper. However, a lot of principal issues are not solved there. First of all, the static mapping mechanisms support only very limited mapping possibilities. Only basic mapping patterns are supported. Mapping and transformation integration possibilities are very limited as well. Each object can be mapped using either transformations or mappings. Mapping definition for ViatraDSM framework has no adequate notation. Solutions to all these issues are the topics of the project described in this paper.

We propose to use a more detailed mapping and transformation integration granularity, for example, to use transformations as preprocessors or postprocessors for mappings. A more expressive mapping language and a mapping definition notation are proposed as well.

There is one more framework GrTP [9] which combines both approaches to a degree, but in a different setting. This framework is based on an advanced

presentation type metamodel, by means of which the desired diagram structure is defined. The framework contains a large set of predefined transformations, which implement all standard user actions related to the defined diagram type. All these predefined actions can be extended or replaced by custom transformations. The main application area for this framework is various conceptual modelling languages; therefore there is no built-in support for domain models. If required, synchronisation with the corresponding domain can be supported by custom transformations, but in future a support for typical mappings to domain could be included.

3 Research Project Description

The main topic of the given research project is how to add mappings to a transformation based tool development framework. The METAclipse framework [6] and model transformation language MOLA built by UL IMCS is chosen as the basis for research project realisation. This choice is based on the fact that the framework is completely transformation based, it provides flexible ways of extension and it itself can be used in a bootstrapping manner for implementing the extended features.

To ensure usability of the proposed approach mappings and transformations should be smoothly integrated. The proposed mapping language could be implemented using an interpreter or a generator generating transformations in a model transformation language (MOLA in our case). This implementation decision affects integration possibilities. In both cases extension points where custom transformations can be added to the functionality defined by mappings could be used. If the generator approach is used we can allow also manual modifications of the generated transformations.

The main extension mechanism should be extension points. For this mechanism to be sufficient in most cases, extension points should be chosen appropriately. Extension points should permit to replace or extend the built-in mapping possibilities by custom transformations.

3.1 The Framework from the User Point of View

The proposed tool definition framework will be metamodel based. At the beginning the domain metamodel of a domain specific language should be built (e.g., by MOLA metamodel editor). The next step would be to define the presentation type model and mappings between the domain metamodel and presentation type model. All this will be done using graphical wizard-style dialogs in the tool development framework.

If built-in mapping possibilities are not suitable for some task, the tool builder will be able to select/create custom MOLA procedure (using the built-in MOLA editor). Appropriate parameters to and from this procedure should be passed, to ensure integrity with the mappings. For each extension point parameters passed to procedures used in this extension point are predefined.

When the tool development is complete, the tool builder can press the button “Build tool”. Thus the tool executable in one step is obtained. Alternatively, if there is such a need the generated transformations can be edited and then compiled.

3.2 Mapping Definition

Mappings are based on typical mapping patterns. A large set of mapping patterns has been identified in Generic Modeling Tool [5] and they will be reused in this project.

Mapping definition is based on the mapping and presentation type metamodels as the abstract syntax of the "mapping language". The visible form of this language will show up as wizard-style dialogs, which will build instances of these metamodels. Appropriate tool support can be built with a small effort using the METAclipse framework. A more detailed description is given in the next section.

Presentation definition in a graphical tool consists of several parts: property dialogs, diagrams as well as model tree, menus etc. Informal mapping examples mentioned so far all have been related to mapping the domain to diagram element types. Now we switch to another part of the presentation – the property dialogs. It is because the proposed ideas can be easier demonstrated on this part and the corresponding metamodels are smaller. In this paper only an essential subset from the property dialog part of the presentation type and mapping metamodels is briefly sketched (in Fig. 2). We assume here that typical Eclipse-style dialogs are used.

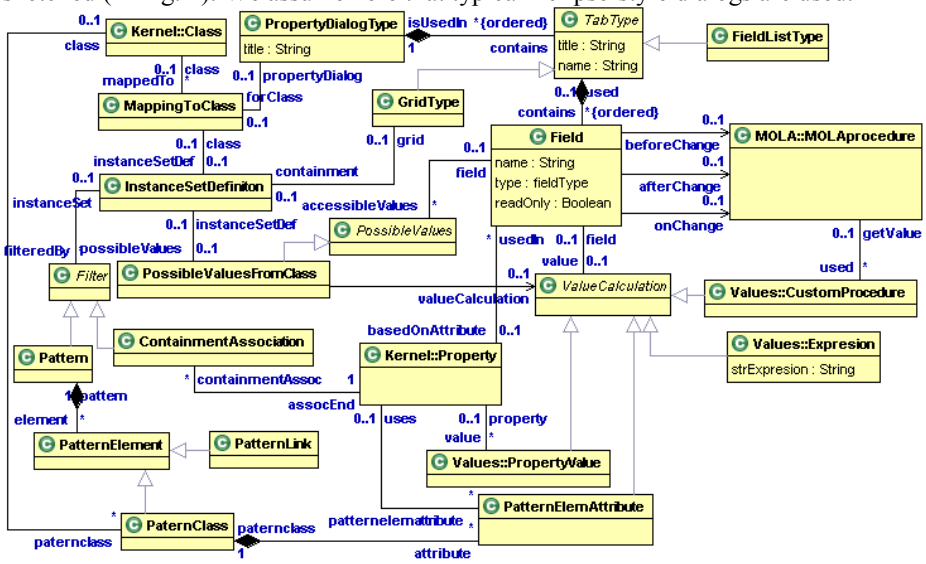

Fig. 2. Mapping and presentation type metamodel subset describing property dialogs

When a property dialog for a domain class is to be defined, at first an appropriate property dialog type (i.e., its structure, element types and functionality) is designed, then it is mapped to domain metamodel elements. A property dialog consists of tabs, which can be either a field list (for displaying class attributes and linked class instances) or a grid (for displaying child instance properties in a tabular form). The basic element of both is a field, whose type definition is the central point in the approach. For each field type it must be defined what must be shown there when the corresponding class instance is selected. For many field kinds (e.g. combobox) the valid value set (e.g., a set of appropriate class instances) must be obtained and

visualized. Finally, it must be defined what has to be done when the value is modified (in Eclipse-style dialogs the model update follows immediately).

As the metamodel in Fig. 2 shows, for all these situations possible typical cases are defined via mappings to domain metamodel elements (e.g., which class attribute must be visualized in a field in the simplest case, see the fragment in Fig. 3).

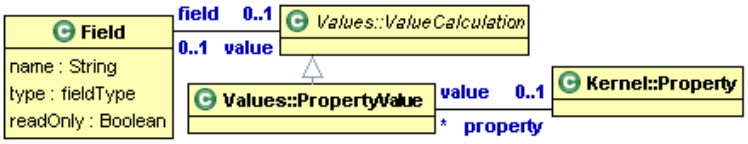

Fig. 3. Metamodel fragment describing design pattern field is based directly on property

The metamodel contains also structuring elements defining various typical ways how these elementary mappings can be combined, e.g., expressions built over elementary mapped values. In all cases the corresponding mapping-based definition can be replaced by a call to a specified custom MOLA procedure. One more novel idea is to use MOLA patterns for defining custom instance set filters, e.g., for selection of relevant child instances.

For example, we can use this mapping language to describe a property editor for UML 2 class diagrams (based on the standard UML 2 metamodel [1]). For UML *Class* a property dialog type could be defined, consisting of two tabs. The first tab will contain a field list describing the UML Class itself. The attributes *name* and *isAbstract* are directly mapped to fields in this tab. For the attribute *name*, a uniqueness check (within a package) before the change is needed, for this task a custom MOLA procedure can be invoked. The second tab could be a grid describing class attributes (see Fig. 4). In this case, the grid *InstanceSetDefiniton* feature is mapped to the *Property* class. The basic instance selection is via *ownedAttribute* master-detail association and additional filtering is defined using MOLA pattern selecting only those properties which are attributes (but not association ends).

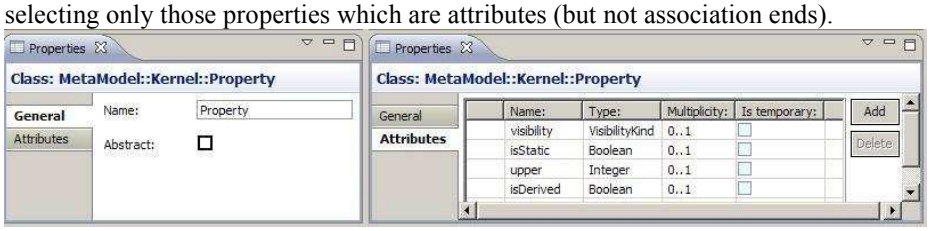

Fig. 4. Class dialog example, general and attribute tab

The metamodel part for the diagram mapping and presentation types can be built the same way, only more classes would be present since it is more complicated.

3.3 Mapping and transformation integration

For the mapping metamodel the most important task is a seamless integration of mappings with custom MOLA procedures. MOLA is a procedural transformation language, therefore MOLA procedures are chosen as the integration unit. It does not

restrict the integration possibilities, since any set of statements can be included in a procedure. Actually it even allows reusing the same procedure in different contexts.
The mapping metamodel granularity and structure should be chosen so that each action could be extended or replaced by an appropriate custom MOLA procedure. The transformation based approach permits to use a more detailed mapping granularity than in traditional mapping based tools.

For each extension point the set of required parameters for custom procedure is predefined. This predefined set should be compatible to the parameter set of the selected procedure.

In Fig. 5 an integration example is given. When a property dialog field is modified a custom transformation can be executed as a preprocessor, postprocessor or instead of the action implied by the static mapping. A custom procedure can be used as well to calculate the field value to be displayed.

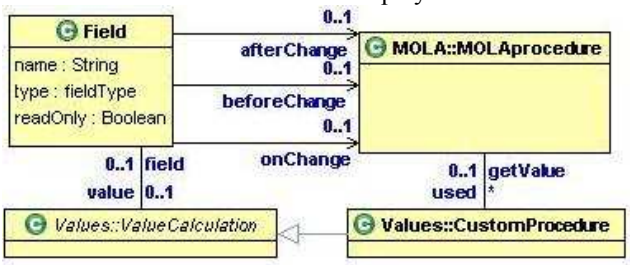

Fig. 5. Metamodel fragment describing mapping and transformation integration

This close integration of mappings and transformation based approach is a key factor in reaching the goal when the transformations generated from mapping only need to be combined with the specified custom MOLA procedures, but require no direct manual modification.

4 Facilities Required to Implement the Approach

To implement the ideas described in the previous section several facilities are required. First of all, a user interface for the mapping definition language should be defined. Then the language implementation is needed. It means an interpreter or code generator for this language is required. This interpreter or generated code should be compatible with the METAclipse framework.

4.1 Mapping Definition Language User Interface

We propose to use wizard style dialogs for the definition of presentation type model and mappings. These wizards will create instances according to the relevant metamodel. The presentation type and mapping definition will be integrated.

To generate presentation types and mapping for a domain class, the user will be asked to select the appropriate tool design pattern and enter additional properties of presentation types to be created (for property dialog, diagram node type etc.). The

relevant mapping instances will be created automatically. The palette element if needed will be created simultaneously as well.

Wizards will be organised in several levels, for the whole domain metamodel (as in GMF) or on one domain class to see or modify the features related only to this class.

In addition to presentation and mapping definition, wizards will allow for complicated cases to select custom MOLA procedures for the relevant extension points. These procedures will be created using the built-in MOLA editor.

A natural way to implement the proposed mapping definition editor in METAclipse framework is to build it as an extension of the existing MOLA tool [6]. Then slightly extended metamodel definition editor can be reused for domain metamodel creation and MOLA editor can be used directly for creating custom procedures.

The mapping/presentation wizard itself could be implemented in several ways. A classical wizard style dialog sequence could be built, but this requires certain extensions to METAclipse property engine. A more interesting and user friendly way could be to create wizard diagrams. The dashboard in GMF [2] could serve as a simple prototype for such diagrams. The possibilities of METAclipse permit to create dynamic wizard diagrams where each node represents some wizard dialog "page". The dialog in such a page can be defined using standard METAclipse property dialog facilities. The edges in such a diagram represent the order in which these pages must be visited. Next nodes and edges will be created and existing ones enabled/disabled in response to the values the user has entered in the current node. A simplified sketch of a wizard diagram for a domain class mapped to node can be seen in Fig.6. It is assumed that the user currently defines tabs for the property dialog.

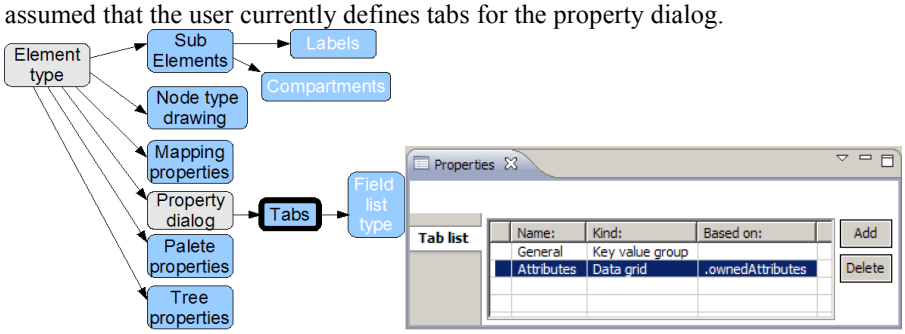

Fig. 6. Wizard diagram example, for domain class mapped to Node.

The same visual representation can be used to modify the defined mappings. After opening the appropriate wizard diagram the user can select a node and update properties. If this modification influences dependencies to other wizard nodes the user is asked to update these nodes as well.

We can think about other mapping visualisation possibilities too. For example, a "mapping diagram" similar to the one in Microsoft DSL Tools [4] can be used, with domain metamodel in one side of the diagram and presentation type model in another, and with mapping lines connecting them. A lot of improvements are possible for this idea. The domain part could be visualised by a standard class diagram. A palette element (if needed) can be shown together with the presentation type. A presentation type can be visualized close to the node with this type. Instead of a label a short form

of the template how this label value will be calculated can be shown. Subelement mappings could be shown in a similar way too.

4.2 Mapping Language Implementation

As it was already mentioned a mapping language implementation is needed and this implementation can be done via an interpreter or generator.

One of the solutions is to create a generator generating MOLA transformations from the defined mappings. MOLA is selected as the target language since it is the base language of the METAclipse framework and custom transformations will also be in it. In addition, it will be easier for the user to modify the generated transformations if needed.

The most straightforward approach would be to define this generator in MOLA language as well. However, a more interesting solution requiring less effort to be implemented can be provided in this project. It is possible to define a "MOLA template language". It will be a template language, combining executable parts in MOLA with templates for transformations to be generated. The first experiments (about 10 % of generator written) show that generator algorithms in this Template MOLA could be defined quite easily. It should be noted, that the Template MOLA has a value of its own as a general purpose macro-processor for MOLA. A more detailed description of this language is given in the next section. One more aspect is that Template MOLA would permit to modify the generator procedures themselves more easily, for example, to adapt the framework to some specific kinds of DSLs.

Another possible way to implement the transition from mapping definitions to MOLA would be to build a universal interpreter in MOLA which would directly interpret them. Some experiments show that the interpreter would consist of procedures quite similar in form to those used in generator. Certainly, some true extensions to MOLA language and compiler would be required in this case. Also, there would be impossibile for tool builder to modify the "generated" code. However, the total effort for interpreter approach could be less.

4.3 Template MOLA

The "MOLA template language" is a direct generalisation of popular textual template languages (of the kind model-to-text) to graphical languages. The planned Template MOLA language would contain two kinds of MOLA statements: standard ones to be executed during the generation process and those to be "copied" to the generated "code" (in fact, model) with template expressions replaced by the appropriate generation time values. Some interesting solutions could appear here, for example, how to generate a set of similar procedures from one template procedure. The template part of the language requires some natural extensions of MOLA syntax, for example, reference to a parameterized class in the MOLA pattern definition.

The metamodel for transformations in Template MOLA consists of three parts. The first one is the generation time part. As already mentioned, there are statements executed in generation time in Template MOLA. These statements are similar to

traditional MOLA and elements used in them should reference the generation time part of the metamodel. In the context of tool building the mapping and presentation type metamodels should be used as this part. Then there is the template part of Template MOLA. Situation with this part is more complicated. There are constant and variable parts of template statements. By constant part we understand elements to be copied directly to the generated MOLA code. Accordingly, there must be also the constant template part of the metamodel. The constant part of template statements should reference only the corresponding part of the metamodel. This metamodel part must be present in the Template MOLA definition environment and must be copied to the runtime metamodel. In the given context, the presentation metamodel is this constant part. There is also the variable part of template MOLA (elements with parameterized types – in angle brackets). For example, the parameterized type "<class>" means that some class should be used here. The generator part during its execution has to substitute this parameter by a reference to an appropriate class. The variable part of metamodel must be provided before the generator is run. In the given context this variable part of metamodel is the domain metamodel (which is supplied by tool builder before the generator execution).

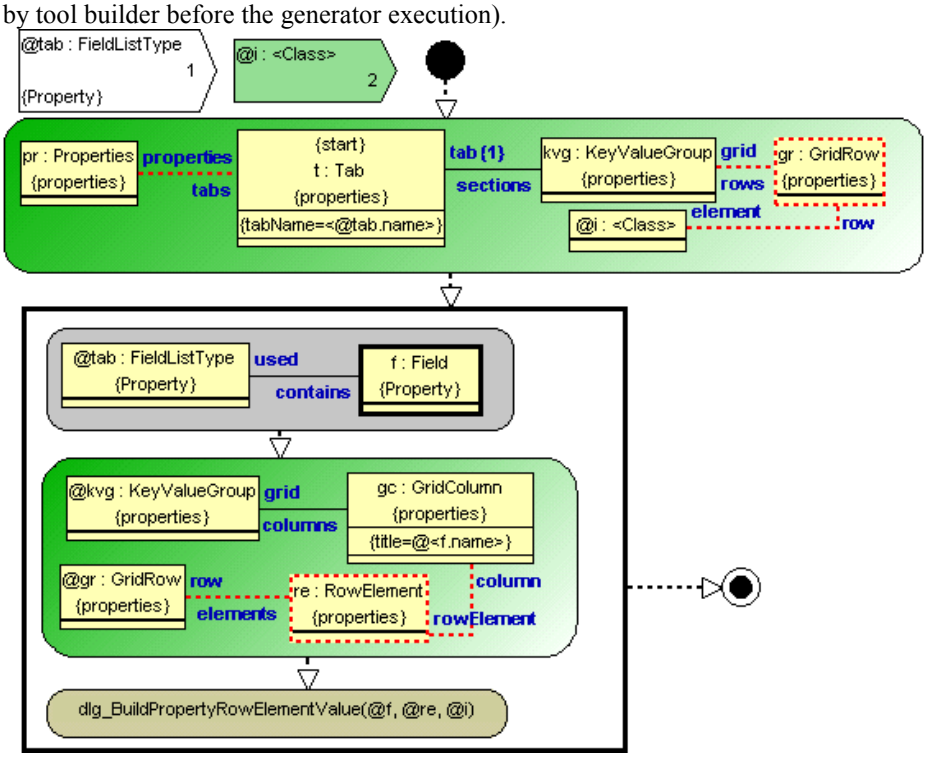

Fig. 7. Template MOLA example

You can see a Template MOLA procedure example in Fig. 7. The procedure generated from this template procedure creates a property dialog tab which actually is of *FieldListType*. We assume that the basic structure of property dialogs is already created during initialization. This procedure is called when some property dialog

should be visualised, its goal is to fill the "prefabricated" dialog structure with data. The first task is to find the already created tab by its name. Then for each field its value is set using the procedure *dlg_BuildPropertyRowElementValue*.

This procedure has one generation parameter *@tab* and one template parameter *@i*, the template parameter has a parameterized type. Another Template MOLA procedure invoking the given one will supply a particular class for this template parameter (in fact, a domain class), therefore in the generated procedure this class will be used as the type of this parameter. The procedure begins with a template rule. Four class elements have types from the presentation metamodel (the fixed part of template metamodel) and one is a reference to the template parameter whose type is substituted the same way as in the parameter itself. During generation this rule is copied to the generated MOLA code, replacing the type of the parameterized element in the described way. Then the procedure contains a loop executable in generation time. It is executed for each *Field* in the *FiledListType*. This loop contains a template rule and a procedure call. In this template rule all elements have types from the fixed part of template metamodel. For each loop iteration one copy of this template rule is created. In the generated code only *@f.name* template expression is replaced with its generation time value. The procedure call means both the generation time call and a generated call, the generated call (to the corresponding generated procedure) will contain only the template parameters. Control flows are generated as well in an appropriate way.

The implementation of template MOLA itself would also not be very complicated. The Template MOLA editor could be built in METAclipse framework using the MOLA editor as basis. The experience in creating MOLA editor shows that it could be done quite easy and would take about one men-month. Also a compiler for Template MOLA is needed. It could be implemented in two steps. The first step would be a "preprocessor" converting template MOLA to traditional MOLA. Certainly, only the abstract syntax form (model) of MOLA can be easily generated, but this is sufficient for the subsequent compilation. In the second step the existing MOLA compiler could be used. The preprocessor converting template MOLA to ordinary MOLA seems to be not very complicated. We have written approximately 10 percent of this preprocessor and these experiments are very promising.

5 Conclusions

The overall goal of this research project is to develop the scientific basis required to create a DSL tool development framework with integrated mapping and transformations support. The main target is to develop language and metamodel facilities for this framework. Only an experimental version of the framework is planed to validate the proposed approach and languages.

Currently draft requirements for such a tool development framework have been developed. The first version of mapping definition and generation/interpretation languages has been developed. These languages should be improved and tested on real life examples. Detailed architecture of the framework should be developed including tool support for the proposed languages.

Actually, ideas and languages described in this paper could be used in terms of other frameworks as well. It would be easier to apply these ideas to transformation based frameworks because their structure is quite similar to METAclipse.

Tool development framework implementing ideas described in this paper could be built with a reasonable effort. It would permit to reduce the transformation size for MOLA Editor approximately by a half. Tool for MOF QVT Relational graphical form [12] (using the original metamodel from OMG) could also be built relatively easy using this framework. The framework would be suitable for building advanced workflow editors as well, for example advanced BPMN implementation with syntax directed editing.

Though there are many open questions, first experiments (redefining some parts of MOLA tool) seem to be very promising.

Acknowledgments. The authors would like to thank Agris Šostaks and Edgars Celms for their help and suggestions.

References

1. Meta-Object Facility (MOF), http://www.omg.org/mof/
2. R. C. Gronback: Eclipse Modeling Project: A Domain-Specific Language Toolkit, Rough Cuts, Addison-Wesley Professional, 2008
3. MetaEdit+, http://www.metacase.com/
4. S. Cook, G. Jones, S. Kent, A. C. Wills: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.
5. E. Celms, A. Kalnins, L. Lace: Diagram definition facilities based on metamodel mappings. OOPSLA'2003, Workshop on DSM, Anaheim, California, USA, October 2003, pp. 23-32
6. A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks, J. Barzdins: Building Tools by Model Transformations in Eclipse. Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyvaskyla University Printing House, 2007, pp. 194–207.
7. C. Ermel, K. Ehrig, G. Taentzer, E. Weiss: Object Oriented and Rule-based Design of Visual Languages using Tiger. Proceedings of GraBaTs'06, 2006, pp. 12
8. I. Rath, D. Varro Challenges for advanced domain-specific modeling frameworks. Proc. of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France.
9 J. Barzdins, A. Zarins, K. Cerans, et. al. *GrTP: Transformation Based Graphical Tool Building Platform*, Proc. of Workshop on MDDAUI, MODELS 2007, Nashville, USA.
10. A. Kalnins, J. Barzdins, E. Celms: Model Transformation Language MOLA. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62-76
11. UL IMCS, MOLA pages, http://mola.mii.lu.lv/
12. MOF QVT Final Adopted Specification, OMG, document ptc/08-04-03, 2008.
13. G. Taentzer, A. Crema, R. Schmutzler, and C. Ermel Generating Domain-Specific Model Editors with Complex Editing Commands. In Proc. AGTIVE 2007, Universität Kassel, Germany, October 2007.
14. 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