SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 77–98 P.

Transformation Synthesis Language – Template MOLA

Elina Kalnina, Audris Kalnins, Edgars Celms, Agris Sostaks, Janis Iraids
University of Latvia, IMCS, Raina bulvaris 29, LV-1459 Riga, Latvia
Elina.Kalnina@lumii.lv, Audris.Kalnins@lumii.lv, Edgars.Celms@lumii.lv,
Agris.Sostaks@lumii.lv, Janis.Iraids@lumii.lv

Higher-Order Transformations (HOTs) have become an important support of the development of model transformations in various transformation languages. Most frequently HOTs are used to synthesize transformations from different kinds of models, for example, mapping models. This means that model-driven development (MDD) is successfully applied to transformations as well. The standard HOT solution is to create the transformation as a model using abstract syntax. However, for graphical transformation languages, a significantly more efficient solution would be to create the transformation using its graphical (concrete) syntax. An analogy here could be the textual template languages such as JET which directly create texts from a model in the concrete syntax of the target language. This paper introduces a new kind of language – a graphical template language for transformation synthesis named Template MOLA. This language is used for creation of transformations in the MOLA transformation language. Template MOLA is an adequate solution for many typical HOT applications.

Keywords: higher order transformations (HOTs), model transformations, template-based language, Template Mola.

1 Introduction

Model-driven development (MDD) has recently become a widespread technology for various kinds of software development. In addition to modeling itself, the key support feature of this technology is model transformations. This has given rise to various model transformation languages, both textual and graphical. We can state that transformation development has become an essential part of software development, with transformation languages being a domain-specific development environment. This domain is characterized by the fact that it itself is well defined by models. Therefore, MDD can be naturally applied to transformation development, i.e., transformations are used to create transformations, as a rule, in the same language. This kind of transformations is named Higher-Order Transformations (HOTs). The idea of HOTs can be applied to virtually any model transformation language. However, the largest number of HOTs important in practice has been created in the ATL language [1], probably due to the fact that the largest known number of transformations has been created in ATL. Automatic creation of transformations from various mappings between two models is especially popular. A large set of such mappings have been obtained by applying the ATLAS Model Weaver (AMW) [2] – a special framework for defining a mapping between two models on the basis of their metamodels. The mappings obtained with AMW can be considered a sort of high-level specification of the required model transformation. However, the idea of obtaining a transformation from a mapping is in no way restricted to AMW and ATL and refers to other transformation languages as well.

A comprehensive survey of HOT applications is given in [3]. They are classified into four types, according to the respective types of input and output models. One of the application types is transformation synthesis. This type is most relevant to the research presented in this paper. Transformation synthesis means transformation generation from different sources of information, including the model mappings mentioned above.

In the HOT approach, transformations must be treated as models conforming to the relevant metamodel. There is such a transformation metamodel for almost all transformation languages. If we want to generate transformations in a transformation language, the metamodel of this language will be the target metamodel of the particular HOT. In [3] synthesis of ATL [1] transformations is considered. An ATL model is created and then extracted as a transformation text (since ATL is a textual transformation language). The same task could be performed for graphical transformation languages, for example, MOLA [4]. A MOLA transformation in abstract syntax (the MOLA transformation model) could be created easily in the same way as the abstract syntax of ATL transformations. The transformation visualisation task is harder since graphical MOLA diagrams have to be created. However, it is also technically feasible. At first a transformation to the corresponding presentation model (graphical diagram) should be executed. Then some auto-layout creation library for graph diagrams should be used. It should be noted that for transformation execution visual representation is not needed. Consequently, for graphical transformation synthesis, MOLA (or ATL) could be used as a HOT. However, a better solution is proposed in this paper.

There are many template-based model-to-text languages. For example, popular languages are JET [5], mof2text [6], Xpand [7], Epsilon Generation Language [8]. The basic application of these languages is to create code (in Java, XML or in any other required language) from the PSM model in the standard MDD process. These languages typically contain facilities to navigate the given model according to its metamodel. However, the main advantage of these languages is the possibility to define the text fragment to be generated by the given rule as a textual template in the relevant concrete syntax (Java, XML or any other). The constant parts are fully defined by the template itself. The variable parts in the text to be generated are specified by means of template expressions which typically contain model class attributes and auxiliary variables. These languages have confirmed their practical applicability in code generation for several years.

Besides the approach described above, an ATL transformation text could also be created using some template-based model-to-text language. Since MOLA is a graphical transformation language and fundamentally model-based, textual template languages cannot be applied here. In this paper we address the problem of MOLA transformation synthesis using template-based mechanisms.

A new graphical template-based language Template MOLA for MOLA transformation synthesis is proposed in this paper. In this language elements to be created in MOLA can be defined explicitly in syntax close to traditional MOLA statements. The generation logic in Template MOLA is described by traditional MOLA facilities. This part of the description is executed during the generation process. The elements to be placed in the created transformation are described in a MOLA extension consisting of template statements. This extension again is similar to traditional MOLA, but with a possibility to incorporate template expressions as well. During generation, these expressions are

replaced by the corresponding generation time values based on the elements of the source model. Thus, the idea of textual template languages is leveraged to a graphical language. The main advantages of the template approach are retained – adequate facilities to process and navigate the source model and concrete syntax-based descriptions of elements to be created as a result. The proposed solution is shown to be significantly more convenient for transformation generation than pure use of MOLA as a HOT.

A short description of MOLA is given in Sub-section 2.1. Sub-section 2.2 describes Template MOLA in general. Section 3 describes the metamodelling aspects of Template MOLA. Section 4 describes Template MOLA in detail. Section 5 outlines general implementation principles.

2 A General Description

The Template MOLA language is an adaption of template mechanisms used for textual template languages (of the model-to-text kind) to a graphical language. It is based on the model transformation language MOLA. Template MOLA is used for easy generation of transformations in MOLA from various input models – as a substitute for the classical HOT approach.

All MOLA elements are retained in Template MOLA. Additionally, special template elements for easy MOLA transformation synthesis are included. With them it is possible to define explicitly in a graphical syntax which MOLA elements should be created.

Because of this close integration of MOLA and Template MOLA, we start this section with a short MOLA description. We continue with a description of basic Template MOLA concepts.

2.1 MOLA

MOLA [4] is a graphical transformation language developed at the University of Latvia. It is based on traditional concepts of transformation languages: pattern matching and rules defining how the matched pattern elements should be transformed. The formal description of MOLA as well as a MOLA tool can be downloaded at [9].

A MOLA program transforms an instance of a source metamodel into an instance of a target metamodel. The two metamodels are specified using the EMOF compliant metamodelling language (MOLA MOF). These metamodels, which may also coincide, are both part of a transformation program in MOLA. Mapping associations may be added to link the corresponding classes in source and target metamodels.

MOLA is the model transformation language which combines the imperative (procedural) programming style with declarative means of pattern specification. A transformation written in MOLA consists of several *MOLA procedures* where one of them is *the main.* An example of a MOLA procedure is given in Fig. 1. The execution of a MOLA program starts with the main procedure. Procedures in MOLA may be called from the body of another procedure using *call statements*. Like in most transformation languages, class instances, primitive and enumeration-typed variables can be passed on to the called procedures as parameters. There are other types of statements in MOLA as well, i.e. *rule, foreach loop, text statement,* etc. The execution of a MOLA procedure starts with the *start statement.* The next statement to be executed is determined by the outgoing control flow.

The rule in MOLA represents the classical branching (*if-then-else*) construct of imperative programming. A rule contains a declarative pattern that specifies instances of which classes must be selected and how they must be linked. Only the first valid pattern match is considered. The action part of a rule specifies which matched instances must be changed and what new instances must be created. The instances to be included in the search or to be created are specified using *class elements* in the MOLA rule. The traditional UML instance notation (instance_name:class_name) is used to identify a particular class element and specify the class the instance must belong to. Class elements included in a pattern may have attribute constraints – simple OCL-like expressions. Expressions are also used to assign values to variables and attributes of class instances. Additionally, the rule contains association links between class elements. A class element may represent an instance, matched previously by another pattern. Such class element is called a reference class element and is specified using the name of the referenced class element, prefixed with "@" symbol.

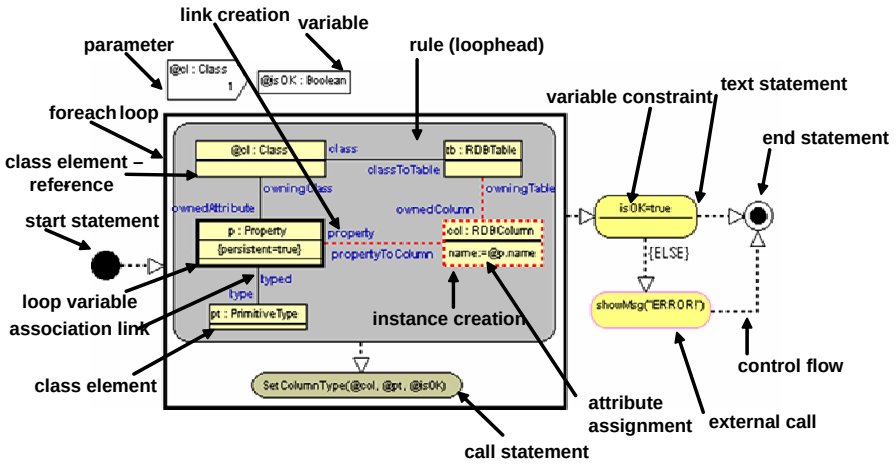

Fig. 1. A MOLA diagram example. The loop is executed over all Property instances which have Primitive Type and belong to referenced Class instance if it is already mapped to an RDBTable

Typical transformation algorithms require iteration through a set of the instances satisfying given constraints. In order to accomplish this task, MOLA provides the *foreach loop* statement. The *loophead* is a special kind of rule used to specify a set of instances to be iterated in the foreach loop. The pattern of the loophead is given using the same pattern mechanism used by an ordinary rule, but with an additional important construct. It is the *loop variable* – the class element that determines the execution of the loop. The foreach loop is executed for each distinct instance that corresponds to the loop variable and satisfies the constraints of the pattern. In fact, the loop variable plays the same role as an iterator in classical programming languages.

The above example demonstrated the concrete graphical syntax of MOLA. The MOLA language also has an abstract syntax defined by means of a metamodel containing several packages (see the MOLA reference manual available in [9]). The abstract syntax of MOLA MOF is defined in the *Kernel* package, with elements *Class*,

Type, *Property*, *Association* and others (actually, it is a subset of a UML 2 class diagram metamodel). The abstract syntax of MOLA procedures is defined in the *MOLA* package (containing *Rule*, *ClassElement*, *AssocLink*, etc). In the next sections, this abstract syntax is referenced where necessary, for example, *Kernel::Class* means a metamodel (MOLA MOF) class.

2.2 Template MOLA

In this sub-section, the basic constructs of Template MOLA are described. The proposed Template MOLA language contains two kinds of MOLA statements: generation statements and template statements.

Generation statements are executed during the transformation generation process. They are used to define the logic of generation process on the basis of the provided input metamodel. All ordinary MOLA statements may be used as the generation statements.

Template statements are meant to be "copied" to the generated "MOLA code" (in fact, model) with template expressions replaced by the appropriate generation time values. Template statements look similar to ordinary MOLA statements but can be distinguished by their graphical style – green color. The most used template statements are template rule and template loop; however, other MOLA statements may be used as template statements too.

Statements in Template MOLA are organized into procedures in the same way as in the traditional MOLA described in the previous section. A procedure may contain both generation and template statements; however, the generation statements alone should constitute a valid MOLA procedure. Template statements may be interspersed between generation statements. Thus, the general idea of Template MOLA is that the "generation part" of a procedure is executed in the same way as the traditional MOLA. The only difference is that template statements to be executed in this process are copied to the resulting traditional MOLA procedures (instead of directly executing them). Certainly, there are some more complex situations to be described further, but at the first glance, Template MOLA means exactly that.

The most used template statement is *template rule*. In generation time it is copied to the generated "code" (i.e., to the relevant generated MOLA procedure). Elements of the template rule may contain variable textual parts – *template expressions* (expressions enclosed in angle brackets followed (preceded) by a percent sign). These expressions are replaced by the corresponding generation time values.

Example of a template rule can be seen in Fig. 2. In this rule, the constraint in class element *b:Class2* contains the template expression *<%@p.name%>* where *@p* is a known generation time reference (defined in the procedure containing this rule). Another kind of a variable part in a rule is a template expression specifying the class of a class element (here *c:<%@tc:Class%>*). The generation time reference *@tc* must point to an appropriate metamodel class, i.e., it must point to an instance of *Kernel::Class* (the *::Class* suffix in the syntax emphasizes that), and it must be set before the rule under discussion is to be executed. In the resulting traditional MOLA rule, this template expression is replaced by the referenced class name. Association links may also be specified by a template expression in order to adapt to a variable class element at the end. This template expression (*<%@assoc:Association%>* in Fig. 2) must reference an association in the metamodel. The value of this reference must certainly be set correctly

during the generation; in the presented example only the association linking classes *Class2* and *Class3* is valid. In the generated rule, the standard MOLA notation for association links (both role names) is used.

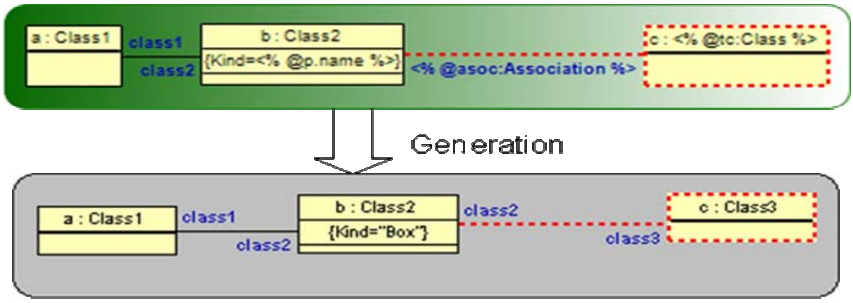

Fig. 2. An example of a template rule and the MOLA rule generated from it

The lower part of Fig. 2 shows the generated MOLA rule obtained from the template rule above. Here we assume that the reference *@p.name* has a string value "Box", the reference *@tc* points to the class *Class3* and *@assoc* to the association with role names (*class2*, *class3*).

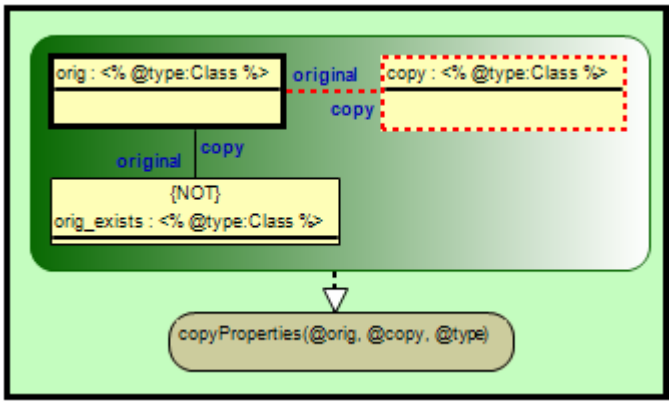

Fig. 3. An example of a template loop

Similarly to rules, the loop constructed in MOLA – the *foreach* loop statement – also has its template form in Template MOLA. The *template loop* is copied to the generated procedure during the generation process, including its body (which may also contain generation statements, see an example in Section 4). The template loop in its *loophead* rule can use all the extensions introduced for the template rule. Fig. 3 shows an example of a template loop, a simple construct for creating copies of all instances of an arbitrary class. In the loophead of this loop, the class to be used in all class elements (including the loop variable *orig*) is defined by a template expression *<%@type:Class%>* which means that the reference *@type* must be set to the required class before the given template loop.

Then a traditional MOLA loop is generated from this template loop, and the generated loop performs instance copying for the given class. The additional class element *orig_exists* with NOT constraint is used as a NAC (negative application condition) preventing from copying the copies again. The example presents a very simple case of another area of typical application of HOTs for transformation generation in [3] – building a generic transformation for a previously unknown metamodel.

The body of the loop in Fig. 3 contains another template-related construct – a MOLA procedure call with arguments of previously unknown types (*@orig* and *@copy*). The type of these arguments becomes known only during the generation process. The given procedure call contains one more argument – the reference to type itself. This last argument is a generation-time argument, which is not included in the generated invocation (it has no sense in that context). Yet for the generation of the procedure *copyProperties*, which has to perform copying of all attributes of arbitrary class, such a parameter could be of high value to define an appropriate generation time loop (traversing the attributes).

The exact kind of procedure parameters is visible in its declaration. There are three types of parameters that can be declared in a Template MOLA procedure – *template*, *generation* and *type* parameters. Template parameters are created in a generated procedure. Generation parameters are used in the generation time and are not created in a generated procedure. Appropriate arguments must be passed in call statements for the template and generation parameters. The type parameters are also used in generation time, but they are inferred from other parameters instead of passing them explicitly. Since the types of parameters in MOLA are described using class *Kernel::Type*, *type* parameters may refer to instances of *Kernel::Type* (*Class*, *PrimitiveType* or *Enumeration*) only.

We have already given an insight into template expressions used in Template MOLA; however, the example does not cover all possible use cases. Therefore, a short summary on template expressions follows. The most common elements where template expressions appear are class elements within a template rule. A template expression can be used to specify the class of the class element. In this case, the template expression must be a reference to *Kernel::Class* instance. If template expressions are used to specify the name of the class element, constraint or expressions in assignment, a string expression is used for this purpose. These expressions may contain generation time variables, parameters and attribute specifications, but not template element references. References to instances of appropriate classes can be used to specify the attribute to be assigned in a class element (a reference to *Kernel::Property*) and the association of an association link (reference to *Kernel::Association*). Template expressions can also be used in template text statements and in call statements to specify arguments which conform to template parameters of the called procedure.

The usage of template procedures in general is more widely discussed in Section 4. On the whole, the idea of generating template procedures in Template MOLA and providing appropriate naming conventions for them is based on principles similar to those in OOP languages such as C++ and Java, also containing some template mechanisms.

2.3 Template MOLA Compared to MOLA as a HOT

A question may arise for the reader, why is transformation synthesis in Template MOLA better than in traditional MOLA? Writing higher-order transformations for

transformation synthesis directly in MOLA requires to define creation of all MOLA metamodel elements explicitly (i.e., according to the abstract syntax of MOLA). To create one rule, we have to create the rule, all its class elements, all association links, all their sub-elements, and to map them to appropriate types from the metamodel of this transformation. Fig. 4 demonstrates a transformation for creation of one rule using traditional MOLA as a HOT language. Creation of the same rule in Template MOLA was demonstrated in Fig. 2.

It is easy to see that the code for creation of this rule in Template MOLA is significantly more readable than in traditional MOLA. Firstly, the size of the rule creation pattern differs significantly. Note that in this example we considered creation of a very simple rule. For more complicated rules, the difference is even more significant. The same situation holds for loops since they mainly consist of rules.

The same issue of complexity arises in regard to other transformation languages also usable for HOT tasks.

Template MOLA allows to implement the same HOT tasks with much less effort and with smaller amount of errors since the structure of the resulting MOLA statements is clearly visible already in the templates.

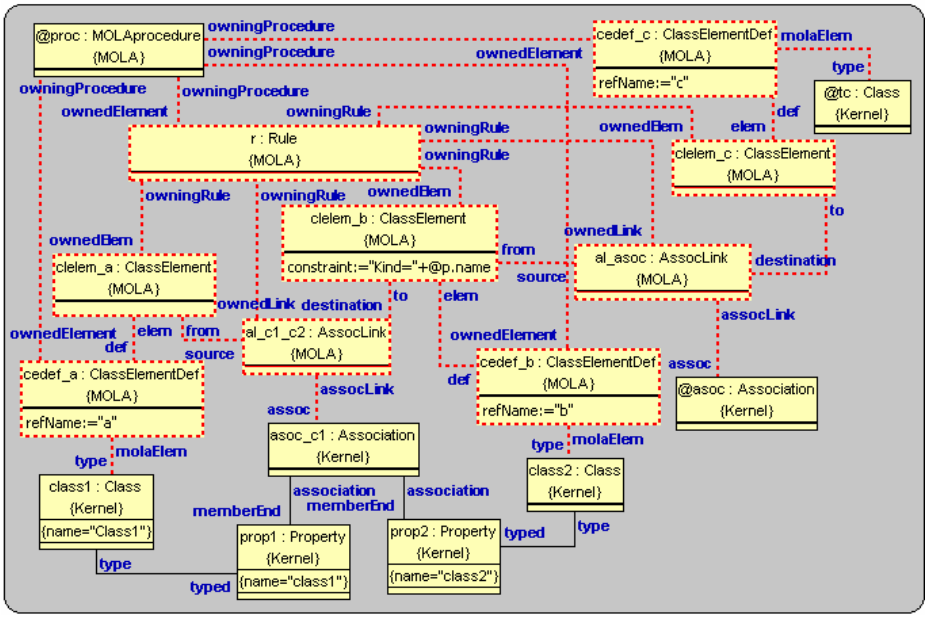

Fig. 4. Creation of the rule from Fig. 2 using MOLA as a HOT

3 Metamodelling Issues

As in any other transformation language, transformations in MOLA are based on the appropriate metamodel definition, frequently containing the source and target part. The definition of a metamodel for Template MOLA is more complicated because the relevant HOT level features for defining the generation logic have to be supported. At

the same time, the use of template statements requires that appropriate parts are present in the metamodel.

In order to have a deeper understanding of metamodelling issues in Template MOLA, we start with the comparison to the metamodel structure required for defining a traditional HOT in MOLA for synthesis of a MOLA transformation (an example of which was shown in Sub-section 2.3). Fig. 5 shows this metamodel structure. The source of the HOT is the source model (a mapping definition or something similar) corresponding to the source metamodel. The HOT must create a complete MOLA transformation definition consisting of a specific metamodel for this transformation (frequently containing the source and target parts) and the proper transformation (a set of MOLA procedures). Similarly, at the metamodel level, the definition of HOT is based on two metamodel parts that serve as a target metamodel for this HOT. Firstly, there are MOLA metamodelling facilities named MOLA MOF MM (the *Kernel* package mentioned in 2.1). Secondly, the MOLA procedure metamodel (MOLA MM) is required.

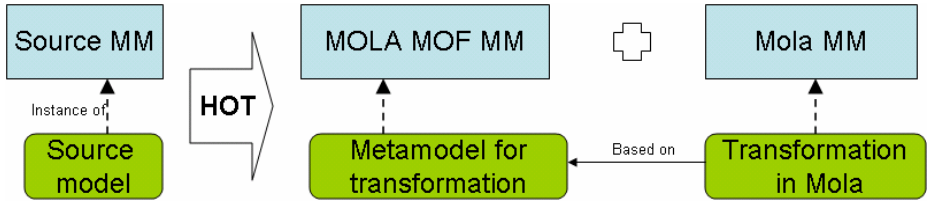

Fig. 5. Models to be used if higher order transformations are written in MOLA

A typical application of HOTs in general and Template MOLA in particular is the generation of transformations from mappings for metamodel-based graphical DSL tool building. The tool building platforms that really require it are METAclipse [10] and ViatraDSM [11]. However, the basic ideas can also be demonstrated in the popular Graphical Modeling Framework (GMF) [12] in Eclipse (we assume for a moment that transformations are generated in MOLA instead of Java for all actions). Fig. 6 illustrates the specialization of the metamodelling situation in Fig. 5, when MOLA transformations are generated by HOT for a DSL tool – i.e., we assume that the GMF generator is implemented as a HOT instead of being written in Java. The source metamodel now consists of several parts with different roles. A definition of DSL normally is based on the relevant domain metamodel (abstract syntax) using, in turn, a version of MOF as a metamodel (in particular, the MOLA MOF could be used in such a role). Another part of the metamodel used by GMF and similar platforms is the presentation type metamodel (named graphical definition metamodel in GMF) and the mapping metamodel. Together they provide the means for graphical syntax definition of a diagram and mapping definition from domain metamodel classes to presentation types in the diagram (by these means instances of the classes must be visualized). The generated transformations in runtime should use the same domain metamodel; therefore, this metamodel must be copied by the HOT to the generated transformation. There also is a constant part of the metamodel – the presentation metamodel (named notation metamodel in GMF) – which defines possible diagram elements at runtime. This constant part also should be created by the HOT. One of the tasks the generated

transformation should do in runtime is to create a visual diagram element for a new domain class instance (according to the defined mapping). Thus, two important special features have appeared in this application: the use of the domain metamodel in two different roles (part of the HOT source and part of the created transformation metamodel), and the constant (independent of the source) presentation metamodel is included in the created transformation. In fact, the reuse of part of the HOT source as a variable part of the metamodel for the created transformation is quite typical when transformations are generated by HOTs from mappings.

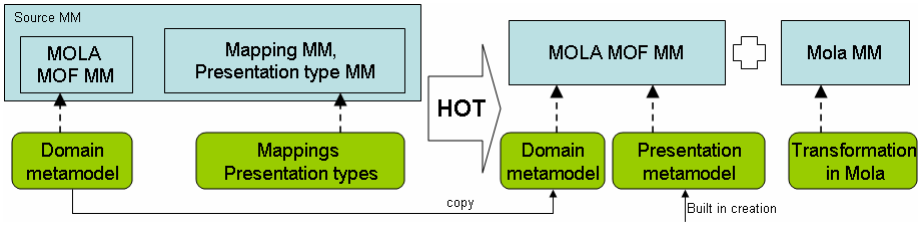

Fig. 6. Models in case MOLA is used as a HOT for tool building

Now we can show what the differences in metamodel structure are if Template MOLA is used instead of a standard HOT approach for the same tasks. Fig. 7 shows the general transformation synthesis by Template MOLA (an analogue of Fig. 5). The "runtime" metamodel for the generated transformation (more precisely, its variable part), as a rule, must also be provided as an input to the Template MOLA-based HOT implementation. This situation could certainly occur in the general case of Fig. 5, but in Fig. 7 this situation is clearly syntactically visible. It is due to the necessity to use template expressions for accessing classes of this variable metamodel part in template rules in a generic way (see Fig. 3). A typical example of such variable part is the domain metamodel for DSL definition (see Fig. 6). What is more different from Fig. 5 is the necessity to provide the constant part of this "runtime" metamodel for the definition of Template MOLA-based HOT. This is due to the fact that classes of this constant part are used to define "constant" class elements in template rules. Therefore, these classes must be defined before the definition of Template MOLA rules. Although this constant part of the metamodel is clearly an instance of MOLA MOF metamodel, in order to be referenced in "constant" Template MOLA elements, it must be provided alongside the MOLA MOF metamodel itself. Metamodel packages included in a complete transformation definition in Template MOLA belong to two adjacent metalevels. However, it is not confusing since the usage of their elements is

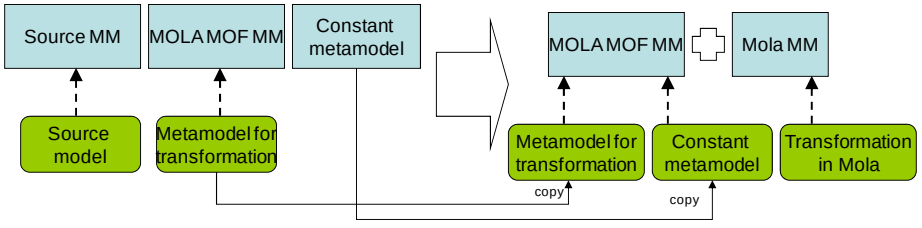

Fig. 7. Metamodels and models used for defining transformations in Template MOLA

clearly distinguished. Note that there may also be elements of another kind in the variable part, but we do not discuss this situation here as it is not typical of our applications.

Finally, we analyse the application-to–metamodel-based tool building in Template MOLA (Fig. 8). The main difference from Fig. 6 is that the presentation metamodel plays the role of the constant part of the metamodel for transformation. Therefore, it must be provided before the definition of Template MOLA. Note that classes for mappings and presentation types can only be used in the generation (non-template) rules and loops of Template MOLA (they play the role of the source metamodel). The domain metamodel is clearly the variable part of the metamodel for transformation. An example of this kind of application is presented in Sub-section 4.1.

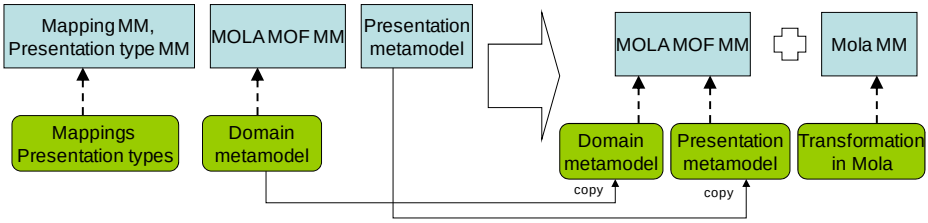

Fig. 8. Metamodels and models used to define transformations in Template MOLA for tool building

Now let us remark on the permitted use of metamodel elements in Template MOLA constructs. Source metamodel elements can be used directly only in generation (non-template) statements of Template MOLA. They can also be used inside template expressions in template statements. Elements of the variable part of the metamodel for transformation (the “runtime” metamodel) can be referenced via corresponding classes of the MOLA MOF in generation statements as well. The same elements can be referenced in template statements only via template expressions for types. The elements of the constant part of the metamodel for transformation can only be used in “constant” class elements in template rules.

4 Template MOLA Examples

In this section, we will demonstrate Template MOLA constructs on examples. In [3] two types of transformation synthesis are considered. We will present an example of each type.

The first is mapping implementation. It means we have some kind of a mapping model to describe dependencies between models. We can generate transformations to implement these mappings from this mapping model. In Sub-section 4.1, a mapping implementation example from the field of tool building will be demonstrated.

The second type is transformation creation for generic metamodels. In this case transformations for a concrete metamodel can be generated from generic transformations. In Sub-section 4.2, we demonstrate how Template MOLA could be applied to such use cases. An example of instance cloning is presented.

4.1 Transformation Synthesis from Mappings

In this sub-section, a simplified example of tool building is presented. Graphical domain-specific languages (DSL) are widely used nowadays. Several tool building environments have been introduced to support tool building for graphical DSLs, for example, GMF [12], MS DSL [13], GrTP [14], METAclipse [10]. In GMF a domain metamodel for DSL is defined in the first step. Then presentation types (in GMF terminology the graphical definition models) and tooling models are defined. Presentation types describe different graphical elements used in the graphical syntax of the language. The tooling models describe palette elements. Then a mapping model that links all these models together is defined. These models are used to generate the JAVA source of the DSL tool. This Java source precisely defines the tool behaviour. Alternatively, a DSL tool-building environment can be transformation-based, i.e., transformations are used to describe the tool behaviour, as it is, for example, in METAclipse. However, there are approaches that combine mappings and transformations [15]. In this case, mappings are used to generate transformations. Since transformation synthesis is needed there, it is a perfect opportunity for application of Template MOLA.

In this section, we use a specific task from the tool-building field as an example. We assume that we have instances of some graphical DSL in abstract syntax (a domain model), and we want to generate the corresponding visualisation (instances of the presentation metamodel). We can certainly write manuallya MOLA transformation, solving the task for this concrete DSL.

In our tool building environment we have means for domain metamodel definition as well as for mapping and presentation type definition; therefore, visualisation transformation for each DSL can be created in a generic way. It means we can build a generic transformation in Template MOLA from which the transformation for visualisation creation in a concrete DSL can be generated automatically.

To write the transformation, we need the corresponding metamodels (built according to the general schema in Fig. 8). A simplified metamodel version is used in this example. The domain metamodel is defined using a small subset of UML (see the upper left side of Fig. 9). Presentation types and a mapping metamodel are also needed. Instances of this metamodel are used as input in the generation time. Here we present a very simple integrated mapping and presentation type metamodel where minimal information on the intended graphical form is included directly in the mapping definition (see Fig. 9, upper right side). Instances of a domain class can be visualised as a box (*ClassToBox*) or as a line (*ClassToLine*). If the class is visualised as a box it may contain several text fields. In these fields, values of some class properties are usually displayed (*PropertyToField*).

During the visualization of classes, the generated transformation has to create instances of a fixed presentation metamodel supported by the tool (see the lower part of Fig. 9). These instances appear only in generated transformations. Therefore, the presentation metamodel is the *constant* part of the metamodel for the generated transformation (compare to Fig. 7 and 8). It describes a graph diagram with *Nodes* and *Edges*. There are *CompositeNodes* containing other *Nodes* and *Labels* for text visualization.

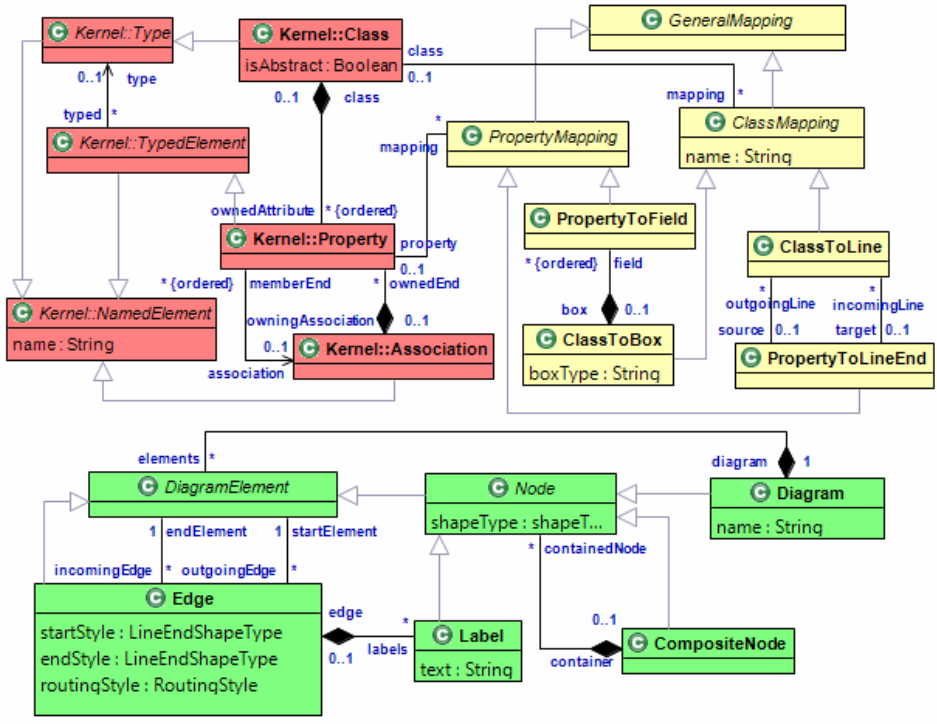

Fig. 9. A simplified domain (upper left side), mapping (upper right side) and presentation (lower part) metamodel

When metamodels and their roles are specified, we can move on to transformation definition in Template MOLA (see Fig. 10). We remind that the proper input for this generation transformation is a specific domain metamodel and a related mapping model. The transformation starts with the loop iterating through all instances of class to box mapping. This loop is a generation loop and is executed in the generation time. As a result, a traditional MOLA procedure is built, containing a loop for each such mapping instance (generated from the template loop which constitutes the body of the generation time loop). The generated loops simply follow each other linked by control flows. The template loop contains the loop variable with the name being generated. The loop variable name is the concatenation of letter *"i"* and the name of appropriate class given by template expression <%@c.name%>. The type of the loop variable is defined by the template expression <%@c:Class%>. In each generated loop the type (@c) is replaced with the concrete domain class corresponding to the mapping instance this loop is generated from. In each loop the value assigned to *shapeType* attribute is explicitly defined. This value is calculated in generation time using the corresponding mapping data (the template expression <%@cm.boxType%> directly references the *boxType* attribute of the current mapping instance). Now in runtime each generated loop iterates over all instances of the corresponding domain class and creates a box for each of them.

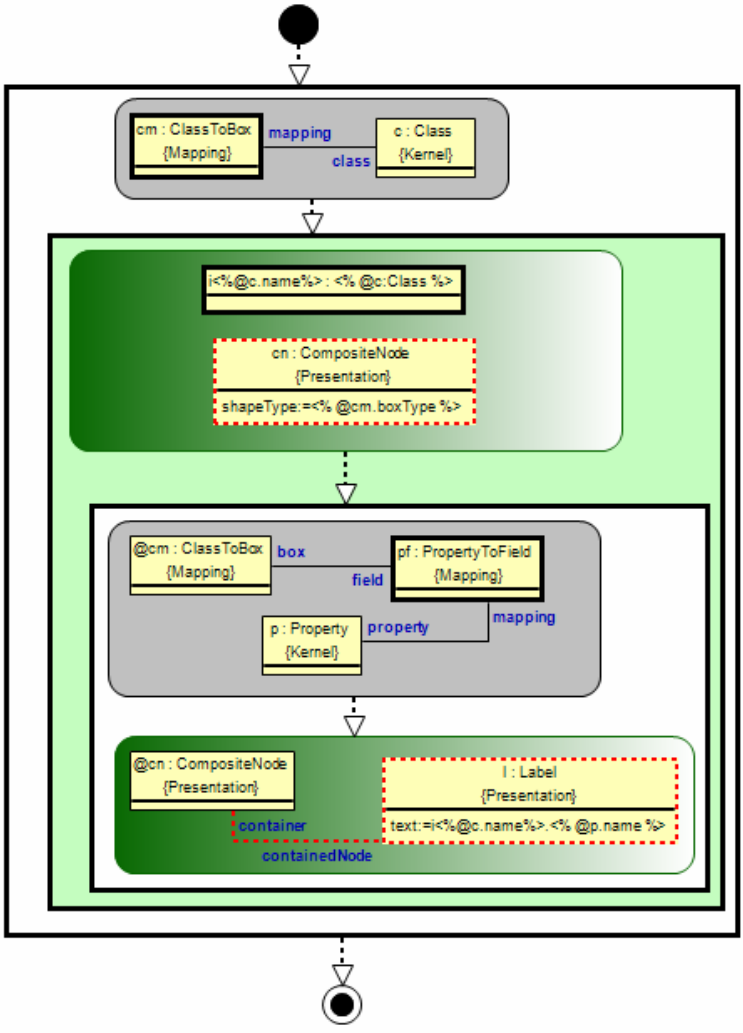

Fig. 10. Mapping implementation for tool building in Template MOLA

We must also generate transformations to create fields and set their values. Therefore, a rule for processing each field has to be generated in the loop body. To ensure this, in the template loop a generation time loop is included. This loop checks which field mappings are included into the given class mapping. For each such field, a rule is created. This rule adds a label to the box and sets its value. To set the value of the label, the relevant property value of the runtime instance should be used. To access this property, the template expression *<%@p.name%>* is used within the assignment in the template rule. During generation the generation time loop ensures that the template expression is replaced with the relevant property each time. It is not difficult to see that the generated sequence of rules will do exactly the required label creation. The structure of the generated procedure is shown in Fig. 11.

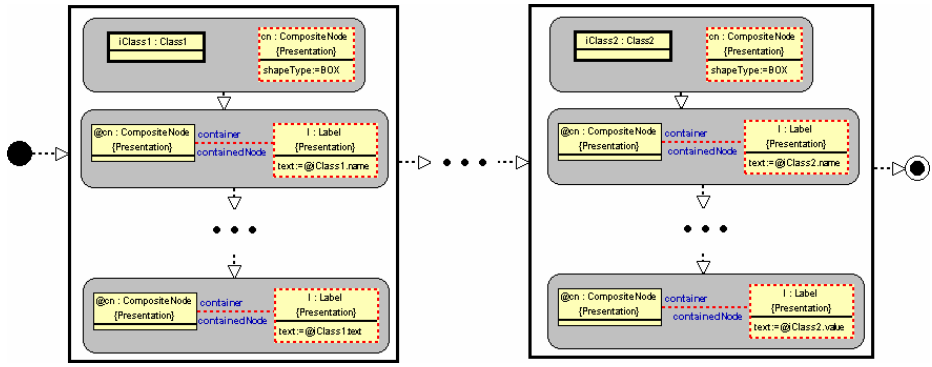

Fig. 11. A MOLA procedure generated for Fig. 10

4.2 Transformations for Generic Metamodels

Template MOLA can be used to write transformations for generic metamodels (the metamodel is unknown at the time of writing). For example, we can write a generic instance cloning procedure. More precisely, in Template MOLA we can write an instance cloning generator, then execute it for a concrete metamodel and run the generated traditional MOLA to clone instances of this metamodel.

Such approach can be used to create reusable transformation libraries. Model transformation reuse has been considered an important topic [16]. One of the obstacles is the complete dependency of transformation definition on the used metamodel. Generic transformations (transformation generators) in Template MOLA could be used to create a reusable library of common metamodel independent algorithms for model processing.

This approach is less important if the transformation language contains features for work with several meta-levels at a time. However, it is useful for transformation languages like MOLA (and most of others that include the OMG standard MOF QVT [17]), which have no support for work with different meta-levels.

Generic Template MOLA procedures can be combined with traditional MOLA. The analogy with C++ templates and Java generics is used here. For example, it is also possible to write such a template based cloning procedure in C++:

```
template <class T> void Clone (T orig, T& copy) {...}.
```

In C++ this template procedure can be called with parameters of a concrete type. To process this template procedure, the preprocessor generates an instance of this procedure for every type it is called with. The same idea is used to combine MOLA with Template MOLA. This feature is required if we want to invoke reusable transformations from a transformation library.

Calls to template procedures can be used in ordinary MOLA transformations. In Fig. 12 calls to the template procedure *Clone* are demonstrated. The same preprocessor technology is kept when combining MOLA with Template MOLA as in C++ when generating procedures for each type they are called with.

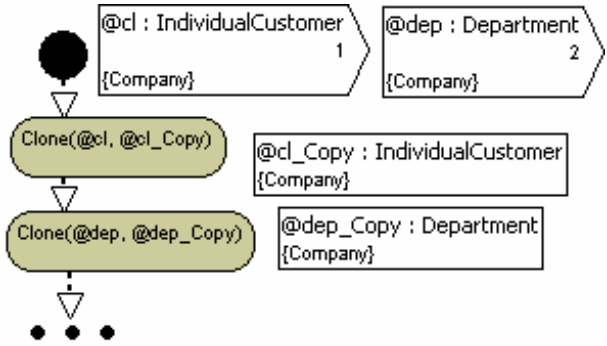

Fig. 12. An example of combining traditional MOLA with Template MOLA. A MOLA procedure calling a template procedure *Clone* from Fig. 13 is shown

Since several MOLA procedures should be generated from one template procedure, the procedure names should be generated too (several procedures with the same name are not allowed in MOLA). For a template procedure, it is possible to define an expression of how procedure name should be generated exactly, but default naming conventions are also provided. One of the preprocessor tasks in combining MOLA and Template MOLA is to replace calls to template procedure with calls to appropriate generated procedures.

Fig. 13 demonstrates the content of the template procedure *Clone*. It contains two template parameters. It means that two parameters will be created in the generated procedure. Instead of type, these parameters contain the template expression *<%@ type:Class%>*. This template expression is evaluated in generation time and replaced with the appropriate values in generated procedures. This procedure contains one more kind of parameter – a *type parameter* (parameter *@type*). This parameter has an analogy to C++ code, where a template parameter *T* was explicitly defined in procedure definition. In the same way as in C++, the value of the parameter is not defined in call but it is inferred from other parameters. Note that this type of parameter is used for this type of transformations only (transformations for generic metamodels) and is not required for typical HOT use cases. Since this template procedure is invoked from ordinary MOLA, the referenced metamodel must be MOLA MOF itself (the *Kernel* package).

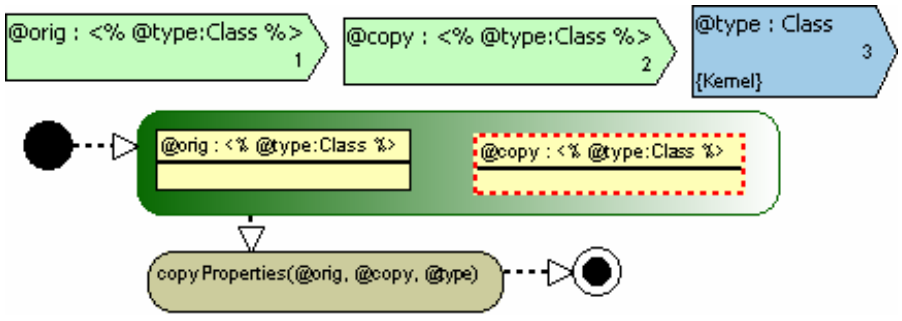

Fig. 13. The *Clone* procedure

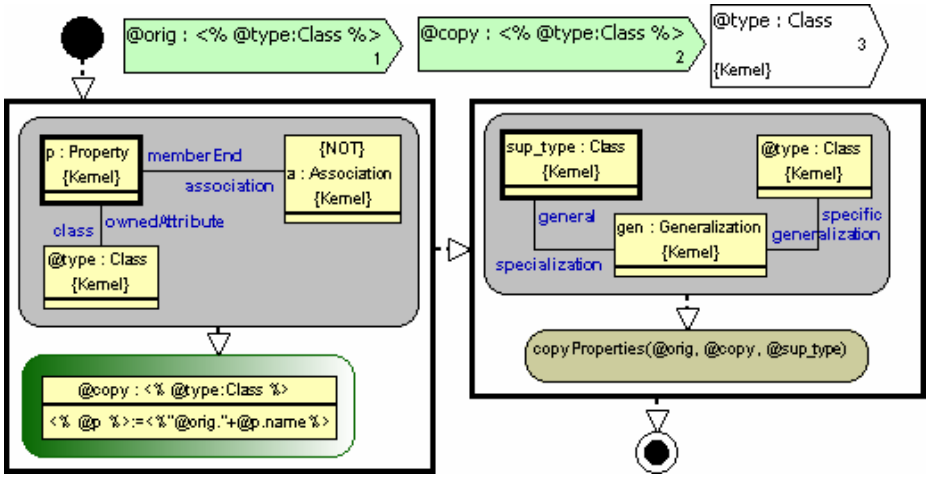

Fig. 14. The *copyProperties* procedure

In the *Clone* procedure one rule and one call is generated. In the rule, the template expressions (which specify types of class elements) are replaced with their generation time values in the same way as in template parameters. The call statement contains one type parameter and two template parameters. The template parameters are kept in the generated call. Actually, instead of a call to the template procedure, a call to the appropriate instance of procedure generated from template procedure is created (taking into account the name generation).

The template procedure in Fig. 14 generates the procedure to copy instance properties. It contains two template parameters and one generation time parameter. The generated procedure will have two parameters created from template parameters. Generation time parameters are only used in generation time.

The *copyProperties* procedure contains two generation time loops. The first loop (on the left in Fig. 14) iterates through all direct attributes of the class. For each attribute, it generates a rule containing a class element with assignment in it. The value of the same attribute in the instance *orig* is assigned to this attribute. In the generated class element, all template expressions are replaced with their values. Template expressions are used for the class element type, for the attribute to be assigned and for the assigned expression. Here is a remark on template expression syntax: the left hand side of the assignment must be an attribute reference in MOLA. Formally, both the notation *@p* (the reference to the attribute) and *@p.name* (a string expression equal to the attribute name) could be used here. Our choice is *@p* since it expresses more directly that the left hand side is a reference (it is preferred for implementation as well).

The second loop (on the right in Fig. 14) iterates trough all immediate superclasses of this class. For each superclass, it generates a call to a procedure that copies direct attributes of this superclass. In this way, using recursion in Template MOLA, values of all attributes are finally copied. It should be noted that the generated MOLA procedures are not recursive due to the fact that procedure names are generated when several MOLA procedures are created from one template procedure. Fig. 16 and 17 explain this situation in an example.

Now let us consider MOLA procedures generated from the Clone algorithm described above using Template MOLA. We will demonstrate the generated result for the first call of the *Clone* procedure in Fig. 12. The type of the instance to be cloned is *Company::IndividualCustomer*. The metamodel for this fragment is described in Fig. 15 (the package containing the fragment is assumed to be *Company*). This could be a simplified metamodel describing the information processed by a company. Fig. 16 presents the code generated form the template procedure *Clone*. The type parameter value is the type of the instance the call statement was invoked with. In this case, it is the class *Company::IndividualCustomer*. In the generated code, the type parameter *@type* is replaced with this class. The procedure call is replaced with a call to the generated procedure with appropriate types. Note that procedure names are generated in Template MOLA as well (according to default name generation rules, which can be modified if required). The procedure name here will be appended by the class name from the type parameter. The procedure name generation is necessary because the generated procedure code depends on the type (or generation) parameter value (as shown in Fig. 17). The type parameter itself is not included in the generated code.

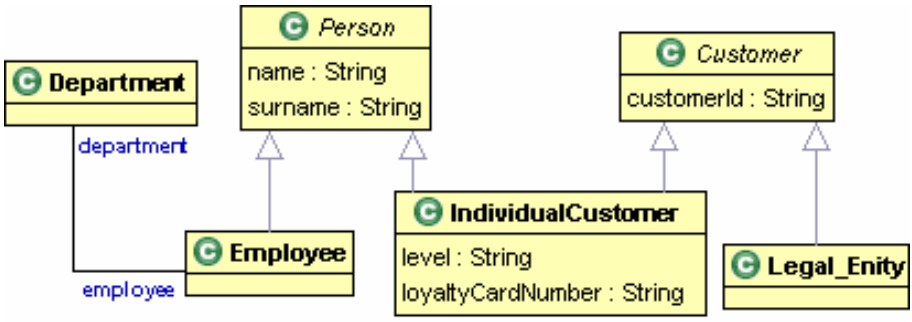

Fig. 15. A metamodel example describing information processed by a company. The class *IndividualCustomer* is used to describe the generated code in Fig. 16 and 17

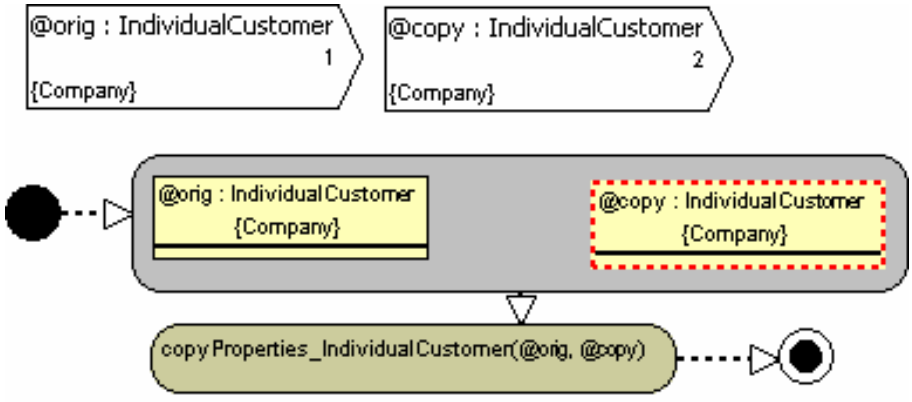

Fig. 16. A MOLA procedure generated from the template procedure *Clone*

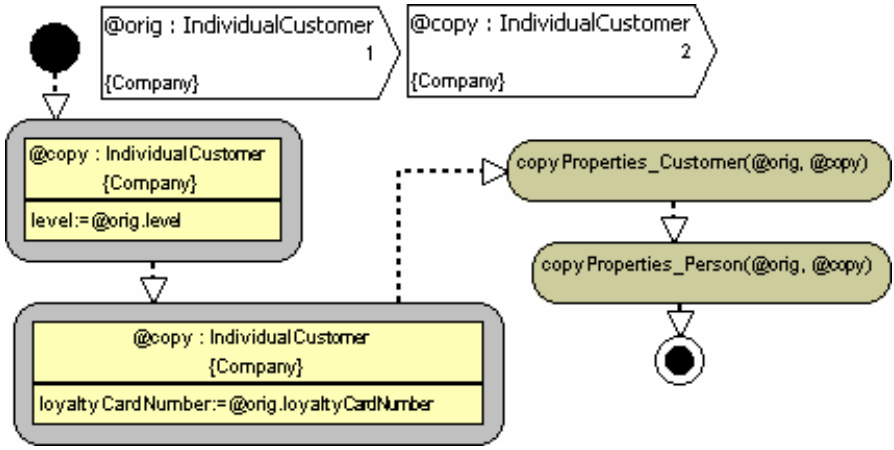

Fig. 17. A MOLA procedure generated from the template procedure *copyProperties*

Fig. 17 presents the structure of a MOLA procedure generated from the *copyProperties* procedure in Fig. 14 when the class specified by the generation time parameter is *Company::IndividualCustomer* (i.e., it is the procedure *copyProperties_IndividualCustomer*). The left side shows two of the generated rules for assigning direct attribute values of the *IndividualCustomer* class (to attributes *level* and *loyaltyCardNumber*). Attribute assignments are followed by calls to copy procedures generated for the superclasses of *IndividualCustomer* (calls for superclasses *Person* and *Customer* are shown). Note that the generated names of the procedures include the class name from the type parameter: thus, there is no recursion in the generated code.

In this example the generated MOLA source is a kind of *spaghetti code*. However, it would be sufficient to have one class element containing assignments for each property. Yet in the current version of Template MOLA, there are no facilities for creation of a variable number of assignments in one class element. This is an open avenue for further research.

5 Implementation Principles

To implement Template MOLA, we have to consider two aspects – editing and processing of Template MOLA.

The Template MOLA Editor was built in a METAclipse framework using the MOLA Editor as a basis. Model transformations which implement the traditional MOLA language within a METAclipse framework have been extended to support the desired functionality in the new editor. Since Template MOLA reuses the syntax from the traditional MOLA language, many of the MOLA procedures implementing the editing actions can be reused. The template elements can be regarded as subclasses of their related “regular” elements, thus inheriting all their required editing behaviour. A template text statement, for example, is almost equivalent to the traditional text statement from the editor’s point of view. New and unique functionality can be easily included where appropriate. So even though a substantial number of new diagram

elements have been introduced, the volume of the code has not grown proportionally, but much less than that. In addition, the subclassing approach eliminates any need for non-trivial migration when converting pure MOLA transformation models to Template MOLA transformation models.

Another aspect is the execution of Template MOLA. Several solutions were considered, including an interpreter and a Template MOLA preprocessor. The chosen solution was to use the preprocessor that converts Template MOLA to traditional MOLA with later reuse of the MOLA compiler to obtain transformations for generation. This approach is similar to preprocessing of macros in C++ environments. The preprocessor replaces Template MOLA statements with traditional MOLA rules which create corresponding instances of MOLA statements. For example, the template rule in Fig. 2 is replaced with the MOLA rule in Fig. 4. The newly-created MOLA transformation is compiled using the compiler of the traditional MOLA language. Finally, the obtained transformation is used as a HOT. Evaluation has shown that the preprocessor solution requires less effort to be implemented.

Another issue for consideration is the readability of MOLA sources generated using Template MOLA. The easiest solution is to create transformations using only the abstract syntax of MOLA. Abstract syntax is enough if we want to execute these transformations without manual extension. However, to obtain concrete graphical syntax for generated transformations, an abstract-to-concrete syntax transformation and an automatic diagram layout generator must be used. Note that transformations in Template MOLA actually contain some layout information for MOLA procedures to be generated. For example, the layout of elements in a template rule could be reused in the generated transformation. However, this issue requires further research.

6 Related Work

The necessity to use Higher-Order Transformations (HOTs) to support many MDD-related tasks was already discussed in the introduction. A comprehensive survey of HOT applications is presented in [3]. Although [3] shows that the classical HOT approach to synthesis of transformations is applicable in practice, it is not always the best solution. Sub-section 2.3 demonstrates how complicated it is to describe creation of a simple MOLA rule directly in MOLA. Creation of transformations in ATL [1] using ATL as a HOT discussed in [3] frequently is similarly difficult. In transformation languages such as Viatra [18], where the metamodelling facilities support simultaneous work at various meta-levels, the usage of HOTs is not required for work with generic metamodels. However, they do not solve transformation synthesis from mappings. In most of other transformation languages, transformation synthesis is even more important.

Therefore, a graphical template language-based solution for transformation synthesis was proposed in this paper. To a great extent, this solution has been inspired by textual template-based model-to-text languages – [5, 6, 7, 8] and many others.

The idea of using a graphical template language for transformation synthesis is new, as far as we know. The comprehensive survey in [19] of various features used for model transformation definition briefly mentions the template-based approach for model-to-model transformations as well. However, the only reference in [20] mentioned

as related to this approach is of templates applied for a very specific task of how to select prefabricated fragments of a target model on the basis of the existence of appropriate elements in the source model.

One more recent approach in transformation development somewhat similar to the approach described in this paper is the use of concrete graphical syntax to define a graph transformation [21, 22]. A graph transformation is defined from the graphical representation of the source model to the graphical representation of the target model. However, the approach is limited and there is no clear application of these ideas to the HOT-related tasks discussed in this paper.

7 Conclusions

A new graphical template based language Template MOLA for MOLA transformation synthesis is proposed in this paper. This language leverages the advantage of template-based model-to-text languages – easy specification of language elements to be generated – on to graphical languages. The graphical template statements of Template MOLA – template rules and template loops – are transferred to the new transformation to be generated. They can contain variable elements – template expressions replaced in the generation process. The generation process itself, which depends on the input model, is defined by means of generation statements – ordinary MOLA statements included in Template MOLA. These generation statements are executed in a standard way during the generation process.

It is shown that it is much easier to specify a transformation synthesis task in Template MOLA than to specify the same task in a traditional HOT style (using MOLA as a HOT).

Several application areas for Template MOLA arise, firstly, metamodel-based tool building for graphical DSLs. More precisely, it is the generation of transformations that determine the tool behaviour according to mappings that define the tool functionality in a static way (as, for example, in GMF). Some research on that has already begun. This paper also provides a small example.

A related application could be generation of transformations from a more general kind of mappings between models. This is the area where HOTs are widely used, especially in ATL.

Another important application is the building of transformations for unknown metamodels. This way, reusable transformation libraries for performing typical model processing tasks could be created. Then transformations from such libraries could be used in ordinary MOLA transformations for a specific metamodel. A very simple example from this area is also provided in this paper.

A future research direction could be to extend Template MOLA for defining templates in other graphical languages, for example, UML activity diagrams. The corresponding template statements then would be defined by the graphical syntax of the generated language. Generation statements controlling the generation process certainly would remain in MOLA. For example, various process generators could be built. This requires more research because implementation could turn out to be more complicated than for Template MOLA.

Acknowledgments. The authors would like to thank Oskars Vilitis for assistance in Template MOLA editor development and Karlis Cerans for valuable comments.

References

1. F. Jouault, I. Kurtev. *Transforming Models with ATL*. Satellite events at the MoDELS 2005 Conference. 2006, pp. 128–138.
2. M. Didonet Del Fabro, J. Bezivin, F. Jouault, E. Breton, G. Gueltas. AMW: a generic model weaver. *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles*, 2005.
3. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bezivin. On the use of higher-order model transformations. *ECMDA-FA 2009*. LNCS, Vol. 5562, Springer-Verlag, 2009, pp. 18–33.
4. A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA. *Proceedings of MDAFA 2004*, Springer LNCS, Vol. 3599, 2005, pp. 62–76.
5. Eclipse, JET. Available: http://www.eclipse.org/modeling/m2t/?project=jet.
6. OMG, MOF Model to Text Transformation Language, v1.0. OMG Document Number: formal/2008-01-16. Available: http://www.omg.org/docs/formal/08-01-16.pdf.
7. Eclipse, Xpand. Available: http://www.eclipse.org/modeling/m2t/?project=xpand.
8. L. M. Rose, R. F. Paige, D. S. Kolovos, F.A.C. Polack. The Epsilon Generation Language. *Proceedings of ECMDA-FA 2008*. Berlin, Germany, 2008.
9. UL IMCS, MOLA pages. Available: http://mola.mii.lu.lv/.
10. 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.
11. I. Rath, D. Varro. Challenges for advanced domain-specific modeling frameworks. *Proceedings of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006*. France, 2006.
12. Eclipse, Graphical Modeling Framework (GMF). Available: http://www.eclipse.org/modeling/gmf.
13. S. Cook, G. Jones, S. Kent, A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.
14. J. Barzdins, A. Zarins, K. Cerans et al. GrTP: Transformation-Based Graphical Tool Building Platform. *Proceedings of Workshop on MDDAUI, MODELS 2007*. Nashville, USA, 2007.
15. E. Kalnina, A. Kalnins. DSL tool development with transformations and static mappings. In: M. R. V. Chaudron (ed.), *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France. Reports and Revised Selected Papers.* LNCS, Programming and Software Engineering, Vol. 5421, 2009, pp. 356–370.
16. J. S. Cuadrado, J. G. Molina. Approaches for Model Transformation Reuse: Factorization and Composition. *Proceedings of ICMT 2008*. LNCS, Vol. 5063. Zürich, Switzerland, 2008, pp. 168–182.
17. MOF QVT Final Adopted Specification, OMG, Document Number: ptc/08-04-03, 2008.
18. Visual Automated Model Transformations (VIATRA2), GMT subproject. Budapest University of Technology and Economics. Available: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html.
19. K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. IBM *Systems Journal*, v. 45 no. 3, July 2006, pp. 621–645.
20. K. Czarnecki, M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Tallinn, Estonia, 2005, pp. 422–437.
21. R. Grønmo, B. Møller-Pedersen, G. K. Olsen. Comparison of Three Model Transformation Languages. *ECMDA-FA 2009*. LNCS, Vol. 5562, 2009. Springer-Verlag, pp. 2–17.
22. J. de Lara, H. Vangheluwe. AToM: a Tool for Multi-formalism and Meta-modelling. *FASE 2002*. LNCS, Vol. 2306, 2002. Springer-Verlag, pp. 174–188.