Baltic J. Modern Computing, Vol. 3 (2015), No. 2, 133-147

Process DSL Transformation by Mappings Using Virtual Functional Views

Lelde LACE, Audris KALNINS, Agris SOSTAKS

Institute of Mathematics and Computer Science, University of Latvia,
Raina bulv. 29, Riga, Latvia

{Lelde.Lace, Audris.Kalnins, Agris.Sostaks}@lumii.lv

Abstract. Every BPMS must have some facilities for transforming its process definition language to the form directly executable by its execution engine. The paper discusses the situation when different domain specific process languages need to be transformed to one target environment. A new approach based on domain specific mappings is proposed. First, the concept of virtual functional view on target metamodel is introduced which permits to describe the creation of target model elements at a higher level of abstraction by simple functions. Then mapping rule definition based on the virtual view is presented which provides in most cases a simple description how source model elements must be passed to these functions.

Keywords: process model transformations, mappings, virtual view

Introduction

Every Business Process Management System (BPMS) is based on a language used for process definition. Currently the most used language for this purpose no doubt is BPMN (WEB, 1). Nevertheless, in cases when processes are quite specific to a business area domain specific process definition languages provide a significant gain in usability. Their main advantage is the possibility to use language concepts specific to the chosen domain, for example, specific action and event kinds with additional parameters. In this sense frameworks where a new domain specific process language (DSL) can be easily defined have the greatest value for process designers.

However, a BPMS has to support the execution of the defined processes, therefore it contains a process execution engine which directly executes processes in some language. For BPMS supporting only a fixed process definition language such as BPMN a custom conversion to the execution language can be used. However for BPMS with several process input languages and especially with DSLs to be defined a more general approach is required. Since languages typically are represented as models in modelling tools the most appropriate solution is to use a sort of model transformations.

Thus, typically there is a source model containing a related set of process definitions in the source language, together with a set of related environment elements such as data models, form definitions etc. On the other hand, the execution engine requires a target model containing only elements directly supported by it. Certainly, general purpose model transformation languages such as MOF QVT (OMG, 2011), ATL (Jouault and

Kurtev, 2005), MOLA (WEB, 2) etc. can be used for such transformation tasks. Nevertheless, this kind of transformation – process to process, but in a different notation – is sufficiently domain specific to be supported by a specific transformation technology. In such a domain specific technology it is possible to hide the transformation tasks common to all process languages and thus to obtain significantly more compact transformations for a specific language.

In this paper a domain specific solution based on a new kind of mappings is proposed. The approach is based on the experience by authors in implementing the platform GraDe3 for support of process DSL (Kalnins et al., 2014). The transformation task in such a framework typically has one fixed target metamodel, but a specific source metamodel has to be used for each DSL and a specific transformation as well. Therefore one of the main goals of the platform has been to ease the transformation development as far as possible. This way the support for a new process DSL could be fully created by domain experts, having only a basic knowledge on modelling and transformations. This is essential because the transformation in fact defines the precise semantics of the DSL to be created.

The proposed approach is based on the assumption that both source and target process languages are based on explicit control flow graphs defining the execution sequence. This assumption leads to a structural similarity of source and target metamodels, thus permitting to separate control flow processing from transforming the nodes.

A new concept – virtual functional view of the target metamodel is proposed, which provides a high-level "process-only" view on this metamodel and permits to hide all technical details. This view contains functions really building the target model elements, with attributes set and links created. Common functions can be defined only once for abstract superclasses. Such view has to be defined only once for the given target metamodel – by platform developers. The transformations for a specific DSL can reference the target elements at a high abstraction level – by just specifying the expressions over the source model providing the data required by the functions. In the result, simple straightforward mappings for the given DSL can be defined, showing only which virtual target metamodel elements have to be built for the given source element type and where to find the data required by the functions. Frequently these mappings just say that for the given source metamodel class a given virtual target metamodel class has to be created. The virtual functional view is described in section 2 and mappings in section 3. The conclusions briefly sketch some completely different applications of this idea.

This paper is an extended version of the paper (Lace et al, 2014) presented at Baltic DB&IS 2014.

1. General principles of the approach

Let's take a look on the transformation definition process for a DSL in Fig.. 1. The source metamodel classes have been denoted by the prefix S# and are called further the S-classes. The target metamodel classes have been denoted by the prefix T# and called T-classes. Source and target metamodels consist of two parts. The first one is the Environment part (the data model, form definitions, user roles). The second is the Process part – facilities to describe the process control flow graph. Typically, the Environment parts of source and target metamodels are very similar. Therefore,

straightforward mappings can be used as a definition of transformation for this part, and it is executed before the more complex transformation of the Process part. We also assume that both metamodels are extended to support explicit source-target traceability and all mappings automatically create this traceability information (and rely on traces already created).

For Process part of the target metamodel the Virtual functional view is built. Every class of the virtual functional view, let's call them V-classes, has been tied to a particular T-class. V-classes contain functions and attributes. Attributes of V-classes are used as parameters for functions. Functions, in turn, can be used to create the extended neighbourhood of the corresponding T-class and to connect it to the Environment part.

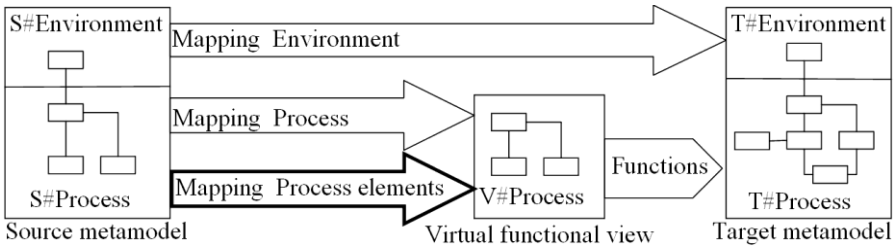

Fig. 1. Definition process of mappings and functions

In the definition of mappings for a concrete DSL the V-classes are used. The functions contained in a V-class are executed when the mapping is applied to a source model. All functions use the attribute values provided in the mapping definition as expressions based on the source model, thus data are transferred from the source model to the target model being built when the mapping is applied. Thus, though a V-class syntactically reminds a normal class its meaning is more like an instance creation rule to be used in a mapping. The mapping has to specify how to find the required attribute values in the source model. Typically, for mapping definitions of different DSL-s only the Process element mappings will differ (the mappings denoted by thicker border in Fig. 1).

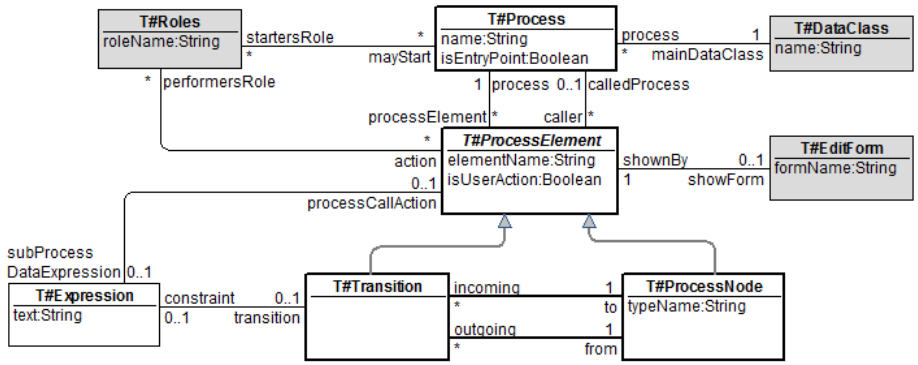

Fig. 2. Target metamodel (execution language metamodel) of GraDe3 platform

We assume that both source and target models are based on explicit control flow graphs. It should be noted that this is true for most of process languages – the standard ones such as BPMN or UML Activity (but not BPEL!) and most of DSLs. And this is true for the execution language metamodel which is used in the platform GraDe3 (Kalnins et al., 2014) and will serve as our target metamodel – see Fig.. 2. Classes with the bold borders correspond directly to the common control flow graph structure. This target metamodel has no explicit sub-classes of T#ProcessNode, the specific action or control node type is coded by the typeName attribute. Environment classes are shown with grey background, and they are assumed to have been already mapped to the corresponding source model elements before we define the process element mapping. Those classes do not directly participate in the control flow, but provide information how exactly the process actions must be executed – which form to open, who can execute the action etc. Certainly, to execute correctly, all required links from process elements to these Environment class instances must be created in the target model. In addition, the Process part of the target metamodel may contain some technical classes, in our case the T#Expression (a class to hold a call parameter expression or the guard condition for a control flow).

The provided target metamodel example in Fig. 2 looks relatively simple, therefore the first impression is that from an adequate source DSL metamodel a set of direct source-target metamodel mapping rules would be easy to define. While it is true in some cases, it is not so for all process elements. First, the Process element part of the source metamodel has more specific classes, in the target metamodel they are just instances of the ProcessNode class with properly set attributes and relevant links to the Environment classes, sometimes several such instances have to be created for one source instance. Further, the real target metamodel typically contains more technical classes with a complicated structure. The found issues are typical for the task – the target metamodel is built with the main goal of supporting an efficient execution engine. One possible solution here would be to introduce an intermediate metamodel containing the same information at a higher abstraction level and in a more structured way. Then it would be possible to build a fixed transformation (mapping) from this metamodel to the target one, and simpler mappings from source to the intermediate model. However, a simple analysis shows that for supporting references all relevant information including the Environment should be stored in the intermediate model as well, thus there would be no significant gain in transformation definition simplicity.

To solve the source-target mismatch in a way really permitting transformations for a specific DSL source model to be developed in a very simple way, this paper proposes to use the virtual functional view as the intermediate representation.

2. Virtual functional view on target metamodel

The virtual functional view is built as a metamodel containing only V-classes and some associations from the T-metamodel. These V-classes in general correspond to T-classes relevant for the control flow, but in a “refactored” way as in building an intermediate metamodel – new subclasses can be added etc., thus giving a more readable and at a higher abstraction level view on the target world. A V-class is used as a target in a mapping, to create an instance of a T-class and set its properties.

A V-class contains a number of attributes and a number of function definitions, but their meaning is slightly different from the traditional class (UML or MOF) notation

used in metamodels. The attributes specify the values which have to be provided in a mapping using the given V-class as a target (these attributes are used as arguments for the functions of this V-class). For each attribute its name and type are specified, the type can be a primitive type, a T-class or a collection of these. In a V-subclass a constant value can be assigned to an attribute defined higher in the inheritance hierarchy.

Each V-class contains a reference to a T-class, specifying that an instance of this T-class is to be created. Functions of the V-class set the properties of this instance – attribute values and links to other T-class instances, on the basis of the provided attribute values. Only non-abstract V-classes are directly used in mapping definitions, the abstract V-classes build the inheritance hierarchy of the virtual functional view. A V-class inherits attributes and functions from its superclasses in a standard way.

Now let us present a simple example of a V-class definition – define the class `V#Process`, referencing the class `T#Process`, without any inheritance hierarchies involved. This V-class causes the creation of a `T#Process` instance when used in a mapping. Fig. 3 shows the simplest part of the definition – the function `setProcessAttributes()` and two attributes used in the function to set the values of the two attributes of `T#Process` class. Both V-attribute values must be set when `V#Process` is used in a mapping, and must have the String and Boolean types respectively. The values are assigned to the two attributes of the new `T#Process` instance (denoted by the keyword self in the function body).

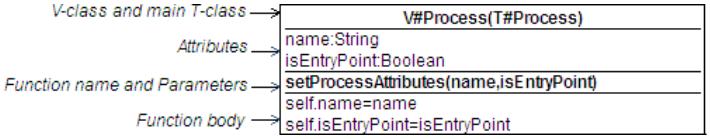

Fig. 3. The attribute setting part of `V#Process`

Next we define a function `connectDataClass()` as part of the `V#Process` – see Fig. 4, the left part. This function performs a typical action in T-model building – create a link between the T-class instance being built by this V-class – the instance named self and an existing target instance, created by a mapping already executed. Here it is a relevant instance of `T#DataClass`. This instance is specified by a new attribute `mainDataClass` of the type `T#DataClass` in this V-class. The attribute is used as a direct instance reference in the link creation.

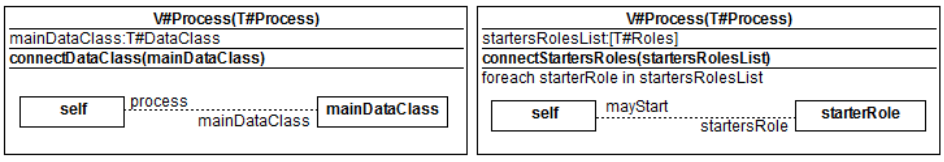

Fig. 4. The connection functions for `V#Process`

A slightly more complicated function `connectStartersRoles()` is defined for creating links (possibly, several) to existing instances of `T#Roles` – see Fig. 4, the right part. Here a new attribute is added as well. But in this case we may need several instances to link to, therefore the attribute type is a collection (list) of elements of the

type `T#Roles` – shown by enclosing the type in square brackets. Accordingly, the function body has to locate all the instances in the list and link them to the self. Therefore an iterator loop construct is provided for the function definition – the iterator runs over all instances in the list, and for each instance (denoted by the iterator variable – `starterRole`) creates the required link.

All three functions used in this V-class are typical ones and represent three function patterns – attribute setting (1) (see Fig. 3), creating a link to a single class instance already in the T-model (2) (see Fig. 4, the left part) and creating several links to existing instances in a loop (3) (see Fig. 4, the right part). Finally, we can show the short form (without function bodies) of the `V#Process` definition – see Fig. 5, the top left corner. One more function pattern 4 is to create a new class instance and link it to self.

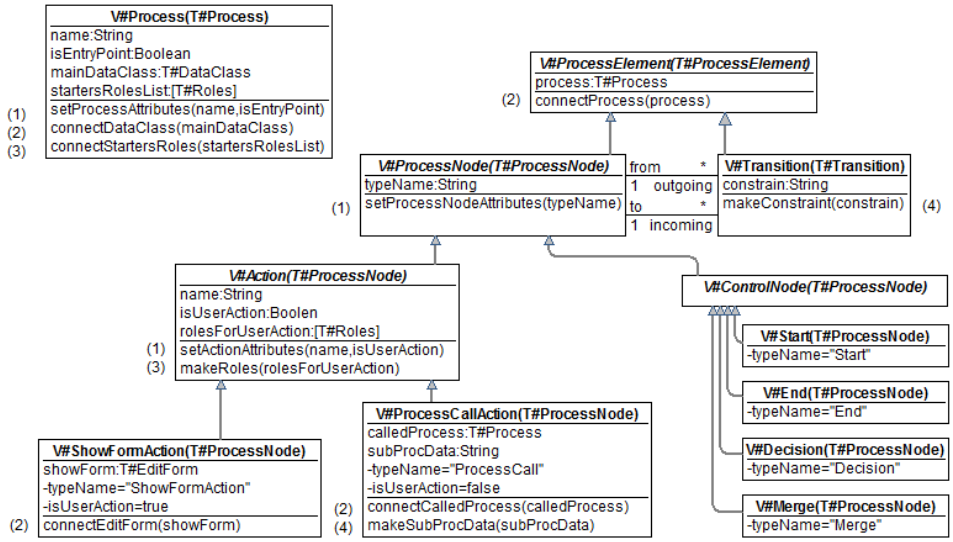

Fig. 5. Short definition of V-classes shown as part of V-metamodel

Next a V-class definition example will be shown involving abstract V-classes and inheritance, this example is based on a fragment of the complete V-metamodel (see Fig. 5) corresponding to our T-metamodel. The goal of this example to define a new V-class `V#ShowFormAction` referencing `T#ProcesNode`. But along the way several abstract V-classes providing the full inheritance hierarchy for `V#ShowFormAction` will be defined as well – `V#ProcessElement`, `V#ProcessNode` and `V#Action`. The first two abstract V-classes correspond to similarly named T-classes, but V#Action is a new V-class introduced to organize in a more clearly way the inheritance of attributes and associations. The definition of `V#ProcessElement` is very simple, with just one function `connectProcess()` of the pattern 2, creating the link to the relevant `T#Process` instance. Similarly, the `V#ProcessNode` definition contains only the attribute setter function (pattern 1). The definition of `V#Action` contains a setter function (pattern 1) for setting the inherited attributes and the function `makeRolesForUserAction()` for creating one or more links to `T#Roles` in a loop (pattern 3).

The non-abstract V-class `V#ShowFormAction` contains one function to be defined – `connectEditForm()`, with the pattern 2, for creating a link to an existing instance of `T#EditForm`. In this V-class constant values are assigned to two inherited attributes (by assignment statements starting with "-"). Fig. 5 shows the short form of these V-class definitions as a V-metamodel, for function definitions their pattern types are shown as well.

Since `V#ShowFormAction` is the only non-abstract V-class in the fragment, only it can appear in mapping rules. Using it in a mapping rule means that all inherited functions will be invoked as well. All attribute and link setting in these functions will be applied to the created `T#ProcessNode` instance. Therefore the attributes of the super-classes (certainly, those not superseded by setting constant values to them) will appear in the list of attributes to be set for `V#ShowFormAction` (however, this attribute setting can be syntactically split into attribute setting rules, see section 3).

We conclude the section with one more V-class required for mapping examples in the next section. The V-class `V#Transition` has one function (with a pattern 4) `makeConstraint()` which creates a new class instance (a `T#Expression`) using a direct string parameter (the constraint text) and link it to the created `T#Transition` – see Fig. 6. A new element is the filter condition (if-expression) which triggers the specified building action only when the condition is true.

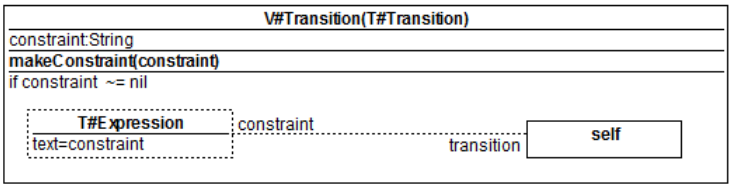

Fig. 6. The function `makeConstraint()` for `V#Transition`.

3. Mappings based on functional view

A mapping definition consists of mapping rule definitions and rule execution order specification. Each mapping rule specifies what target model fragment is to be created for the given S-class. The rule may contain some filtering conditions with respect to instances of this class, but more complicated source patterns are not included – there is no need for them in the domain. The mapping rule body contains one or more V-class occurrences with all attribute values set. Expressions for attribute setting are based on a specific source metamodel, Fig. 7 shows the used here source metamodel corresponding to one specific DSL to be defined. Although the given source metamodel is similar to the target metamodel, it is sufficient to show all basic mapping constructs.

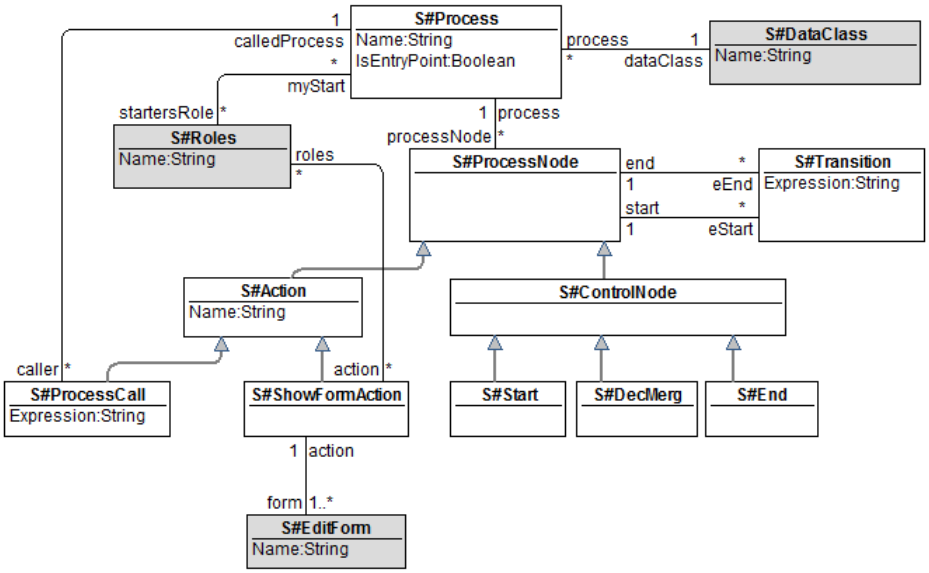

Fig. 7. Source metamodel of an example DSL used for mapping rules.

Mapping rule execution means applying it in a loop to each instance of the specified S-class in the source model and creating T-class instances as specified by the included V-classes. In addition, the standard source-target traceability information at instance level is created. In fact, mapping rules are of several categories and these categories influence their execution order. The categories are listed here according to their execution order.

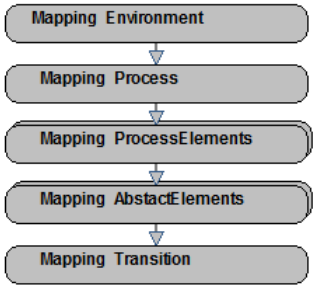

Fig. 8. Mapping execution order specification using categories.

First, there is the Environment category which includes the mappings for Environment classes. The next category is Process, which contains only the rule for `S#Process`. Then the category ProcessElements follows, which contains all "normal" process element kinds in the source model (`S#Start`, `S#ShowFormAction` etc.). These rules create local fragments consisting of `T#ProcessNode` instances – frequently just one instance. But the created fragment may consist of several `T#ProcessNode` instances, if the body contains several V-classes; then internal control flow creation can be defined as well and the fragment must always have one entry and one exit. The execution order of rules in this category is irrelevant. There is also a special category AbstractElements, these rules are based on an abstract S-class and

extend the fragments already created for non-abstract subclass instances of this class. Both the ProcessElelements and AbstractElements rules automatically create an extended traceability – besides the standard source-target traceability links they create links from source instance to the entry and exit of the created T-fragment. The last category – Transition is for building `T#Transition` instances between the created T-fragments.

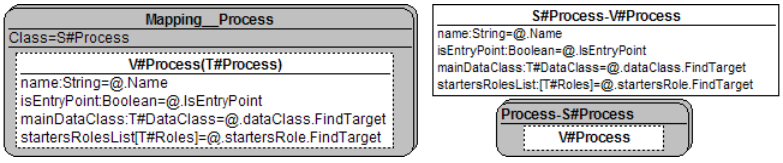

Fig. 9. Example of mapping definition using `V#Process` (two syntax forms)

Fig. 9 (left) shows the simplest mapping rule definition which maps all `S#Process` instances from an S-model built according to the metamodel in Fig. 7 to `T#Process` instances as required – create instances, set attributes, build links. The `V#Process` definition here is even more abridged – only assignments to attributes are visible. The mapping runs over all instances of the specified S-class – `S#Process` and for each does what is specified in the `V#Process` class definition by creating the `T#Process` instance and invoking the functions. The attribute values are set by expressions in the mapping rule on the basis of the current `S#Process` instance, denoted by "@" in these expressions (the standard property-based navigation notation is applied). For class-typed attributes the special built-in function `FindTarget` is applied which returns the instance in the T-model to which the current S-model instance has been mapped (using the traceability information). The `FindTarget` function can also be applied to a list of S-instances, then it returns the corresponding list of T-instances.

Alternative syntax for mapping rules is also available – a rule can be split into a declarative attribute setting rule for a V-class (on the basis of the relevant S-class) and executable T-instance creation rule (see Fig. 9, right). The main advantage of such splitting is that attribute setting rules can be applied to abstract V-classes as well and can be inherited by their subclasses, thus a significantly more compact mapping description can be obtained. This style will be used for the rest of the example.

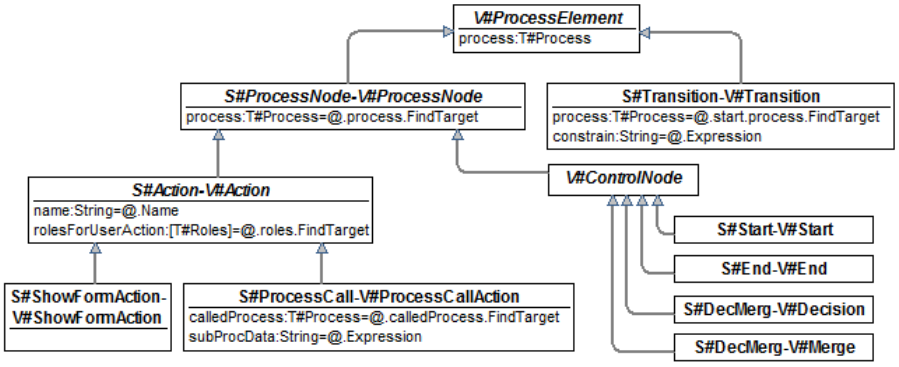

Fig. 10. Attribute setting rules

Now more examples of mapping rules are given. Fig. 10 shows the attribute setting rules for the relevant V-classes in the form of an inheritance tree. There the description compactness is clearly visible, e.g. the attribute specifying the link target from any kind of process node to the process itself has to be set only once in the rule for the `V#ProcessNode`. The attribute rule hierarchy is based on the corresponding V-class hierarchy. Each rule references an S-class; the property navigation in the rule must be consistent with this class in the S-metamodel. These S-classes along an inheritance path in the rule tree must be consistent with the inheritance in the S-metamodel, and end up with a non-abstract S-class for the leaf rule (for a non-abstract V-class which is used in an executable mapping rule). This is because during execution all @-navigation goes from the current instance of this non-abstract S-class. For better readability relevant V-classes without attribute assignments can also be included in the rule inheritance tree – to see what attributes must be set or what V-subclasses may be used.

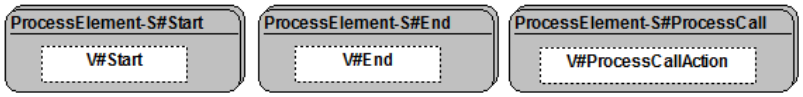

Fig. 11. Mapping rules for `S#Start`, `S#End` and `S#ProcessCall` classes

Now we will describe the executable rules. We begin with some comments on the rules in the category ProcessElement, starting with the simplest ones. All the attribute setting has already been defined in the attribute rules. Therefore rules for S-classes `S#Start`, `S#End` and `S#ProcessCallAction` (see Fig. 11) where only the relevant T-class instance is to be built contain just the reference to the V-class.

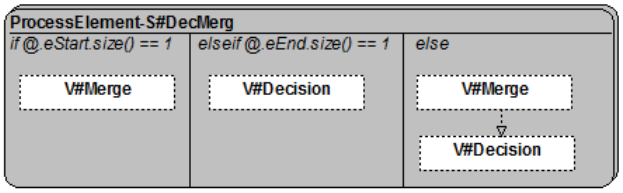

Fig. 12. Mapping rule for `S#DecMerge`

The mapping rule for `S#DecMerg` (Fig. 12) is more complicated, since in our DSL decision and merge control nodes are united into one control node, the meaning of this control node can be deduced from the control flow context – one outgoing control flow means merge, single incoming one means decision, otherwise there must be both merge and decision nodes. To have such case-like functionality, the if-elseif-else construct is introduced. The last case contains two V-classes thus creating two `T#ProcessNode` instances, these instances are connected by a control flow specified graphically.

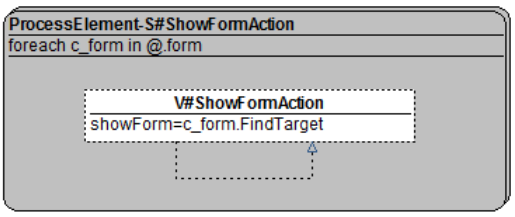

Fig. 13. Mapping rule for S#ShowFormAction

A loop can be defined directly in the mapping, to create a variable number of instances of a "main" T-class. Fig. 13 shows such an example – the mapping rule for `S#ShowFormAction`. According to the metamodel (Fig. 7) in a source model one such action instance is related to one or more forms – the expression `@.form` returns a list of source forms. For each of the forms a separate instance of `T#ProcessNode` has to be created. So the mapping body must be executed for each form instance in this list in the source, the corresponding created target fragments are connected by transitions into a single sequence. Note that this is the only mapping rule where the attribute type for a function in the V-class (`T#EditForm`) doesn't coincide with the expression type required for the mapping rule itself (a list of source forms). Therefore this attribute is not set in an attribute setting rule in the inheritance tree (Fig. 10). Instead, in the mapping rule the value of the attribute `showForm` is set explicitly on the basis of the current source form instance in the list (`c_form`). Certainly, the values of the other inherited attributes are taken from the inheritance tree.

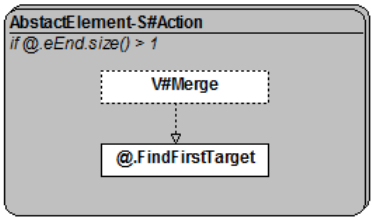

Fig. 14. Mapping rule for the abstract class `S#Action`

Now a rule in the category AbstractElements – the rule for creating an additional merge node (Fig. 14). The DSL permits two or more control flows to enter an action, while the target model requires a merge node to be prefixed to such an action. The rule is based on `S#Action` – a merge node is prefixed to all relevant fragments created from `S#Action` subclass instances (this is done after all such fragments have been created). The new "internal" transition must go to entry node of the existing target fragment; this is specified using the predefined function `FindFirstTarget` (find the fragment entry) based on the extended traceability.

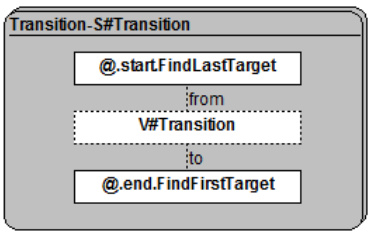

Fig. 15. Mapping rule for S#Transition

We show also the mapping for `Transition` (Fig. 15). The created `T#Transition` must connect the exit and entry nodes respectively of the relevant fragments, the nodes are located by the functions `FindLastTarget` and `FindFirstTarget` respectively. We remind that transition mappings are applied as the last ones, after all fragments for nodes have been finalized. Here two links are directly created as well in the T-model (therefore the corresponding associations from the T-metamodel were included also in the V-metamodel).

We conclude with brief comments on some more possible mapping facilities not shown here. For a loop defined directly in the mapping the corresponding created target fragments were connected by transitions into a single sequence (Fig. 13). However more complicated control patterns such as parallel fan-out – fan-in can also be specified. Completely another dimension in extending the approach is the possibility to define new built-in functions on models, such functions then can be used for various expressions both in V-rule and mapping rule definitions, thus significantly extending the expression power beyond simple model navigation shown here. It seems that these extensions cover most of possible needs for mapping various DSL constructs. However, for practical examples of DSLs built using the platform GraDe3 (Kalnins et al., 2014) the described here facilities were completely sufficient.

4. Related work

The approach proposed in the paper has the largest value for situations when a domain specific process definition language has to be transformed to some executable process language. Especially it is in the case where the corresponding implementation platform supports easy facilities for defining new domain specific process languages including graphical ones. In fact, there are not so much process DSL environments. One of the first such environments is jABC (Steffen et.al., 2007). Some separation of control flow and process element aspects is present there, however only process elements directly implemented as Java classes are considered there, therefore explicit process transformations do not appear. The next to be mentioned is the approach based on Karlsruhe's Integrated Information Management (KIM) (Freudenstein, 2009). There a number of standard process languages (BPMN, UML Activity, Petri nets) are used as source languages, but they are transformed to a common execution language via an intermediate model. Processes in source languages are entered as models, but these models are stored as xml files and the transformations are built in XSLT. Some transformation structuring is provided there, but only via transformation modules

invoked as web services. The approach closest to ours is DSLs4BPM (Heitkötter, 2012), there new graphical DSLs can be defined using the Eclipse framework. All these DSLs have a fixed control structure, but new action kinds can freely be added, and BPMN is used as the target language. Language transformation there is completely model based, and implemented in MOF QVT Operational Mappings language (OMG, 2011). Fixed transformations are provided for the control structure, but new action kinds (building blocks) are transformed by new custom rules to be included via MOF QVTO extension mechanism. Thus some transformation structuring and reuse is provided in DSLs4BPM, but it requires significantly higher transformation development skills than our approach and is less flexible at the same time.

5. Conclusions

A new approach to process transformations by means of virtual functional views on the target metamodel and mapping rules based on such views has been proposed. This approach has been partially implemented in the platform GraDe3 for support of domain specific process languages (Kalnins et al., 2014), with the basic possibilities of mapping rules really present there. However, the concept of virtual view was not explicitly used; the idea appeared only as an appropriate structuring of T-model creation functions, implemented in the low-level transformation language Lua/lQuery (Liepiņš, 2012). The explicit use of V-metamodel would have simplified the transformation development significantly, especially for simple modifications of source DSLs in order to find the most appropriate process DSL for a domain. On the other hand, these experiments have shown that implementation of the whole approach in this environment would not be expensive since the main building blocks are already present.

However, the idea of a virtual functional view of a metamodel could have an even broader applications – in many cases where the target metamodel to be transformed to has a complicated coding of details. Then such details could be hidden in the functions defined in the view. In the result, any creation of target model elements – by a transformation or mapping – could be based on "main" elements of the T-metamodel kept in the V-metamodel and the corresponding functions.

The situation is even quite typical for classical Model Driven Development (MDD) tasks based on UML (OMG, 2010). One such example is UML sequence diagrams where a simple task – to add a new message to a diagram (an arrow representing a class operation invocation) requires adding up to 18 instances of different metamodel classes, thus making transformation definitions overloaded with technical details. In procedural (imperative) model transformation languages such as MOLA (WEB, 2) these problems can be solved to a degree by creating specialized elementary transformation libraries – as in standard programming languages. However it is much more difficult to accomplish this for declarative transformation definition style and mappings – a style much broader used in practice.

An attempt to apply ideas similar to metamodel views for model transformation development in the classical MDD area was made by IMCS team in (Kalnina et al, 2012) where a simplified metamodel view – a tree type definition facility was offered both for source and target metamodels. This way many practical model transformations for UML based MDD could be defined by simple mappings, including sequence diagram creation. However the implementation of this idea required Higher order transformations in

Template MOLA (Kalnina et al, 2010) to be used. In addition, the proposed facilities for creating the target model from the target tree (just extended OCL-like navigation) may be insufficient for more complicated cases, explicit functions would fit there well.

The same problem is important also for the model transformation language most used in practice – ATL (Jouault and Kurtev, 2005). Though ATL supports also the imperative style, most of transformations in ATL are being developed in the declarative style. There matched transformation rules with their *from* and *to* clauses are close to advanced model mappings. For source metamodels there are rich facilities in ATL – helper functions and attributes, which in fact permit to define a required view on the metamodel by extracting the relevant data from the source model in the form most appropriate for use in the given transformation rules. Each helper is defined in the context of a source metamodel class or the transformation module in general. However there are no similar facilities in ATL for accessing the target metamodel in *to*-clauses of rules, all features of the *to*-class instance to be created must be set explicitly in the rule. A setter function concept similar to our functions in V-model could be added to ATL, with the context being a "main" target metamodel class, the environment of which has to be built with all technical details included. The only issue could be the fact that currently all functions in ATL are without side effects.

All this provides a new dimension in transformation structuring, which is yet to be evaluated in practice.

6. Acknowledgments

This paper partially has been supported by the ESF project 2013/0005/1.1.1.2.0/13/APIA/VIAA/049.

References

Freudenstein P. (2009). Web Engineering for Workflow-based Applications: Models, Systems and Methodologies. *Karlsruhe: KIT Scientific Publishing, German*

Heitkötter, H. (2012). A Framework for Creating Domain-specific Process Modeling Languages. *Proceedings of the ICSOFT 2012*, Roma: SciTePress, 127-136.

Jouault, F., Kurtev, I. (2005). Transforming models with the ATL. *Lecture Notes in Computer Science*, Springer, volume 3844, 128–138.

Kalnina E., Kalnins A., Celms E., Sostaks A. (2010). Graphical template language for transformation synthesis. *Proceedings of SLE 2009*, LNCS, Springer, volume 5969, 244-253.

Kalnina E., Kalnins A., Sostaks A., Celms E., Iraids J. (2012) Tree Based Domain-Specific Mapping Languages. *Proceedings of the SOFSEM 2014*, LNCS, Springer, volume 7147, 492-504.

Kalnins A., Lace L., Kalnina E., Sostaks (2014). A. DSL Based Platform for Business Process Management. *Proceedings of the SOFSEM 2014*, LNCS, Springer, volume 8327, 351-362.

Lace L., Kalnins A. and Sostaks A. (2014). Mappings for Process DSL Using Virtual Functional Views, *Proceedings of Baltic DB&IS 2014*, Tallinn University of Technology Press, 371-378.

Liepiņš, R. (2012) Library for model querying: IQuery. *Proceedings of the 12th Workshop on OCL and Textual Modelling (OCL '12).* New York: ACM, 31-36.
OMG (2010). *OMG Unified Modeling Language (OMG UML), Superstructure, version 2.4*
OMG (2011). *OMG MOF 2.0 Query/View/ Transformation (QVT) Specification, version 1.1*
Steffen B., Margaria T.et al. (2007). Model-driven development with the jABC. *Proceedings of the 2nd International Haifa Verification Conference,* LNCS, Springer, volume 4383, 92-108.
WEB (a) BPMN 2.0 specification, `http://www.bpmn.org`
WEB (b) MOLA – MOdel transformation LAnguage, `http://mola.mii.lu.lv`

Received June 9, 2015, accepted June 12, 2015