Scientific Journal of Riga Technical University
Computer Science. Applied Computer Systems
2011
Volume 46

Towards Open Graphical Tool-Building Framework

Edgars Rencis, Janis Barzdins, Sergejs Kozlovics
Institute of Mathematics and Computer Science, University of Latvia, Riga, Latvia

***Abstract* – Nowadays, there are many frameworks for developing domain-specific tools. However, if we want to create a really sophisticated tool with specific functionality requirements, it is not always an easy task to do. Although tool-building platforms offer some means for extending the tool functionality and accessing it from external applications, it usually requires a deep understanding of various technical implementation details. In this paper we try to go one step closer to a really open graphical tool-building framework that would allow both to change the behavior of the tool and to access the tool from the outside easily. We start by defining a specialization of metamodels which is a great and powerful facility itself. Then we go on and show how this can be applied in the field of graphical domain-specific tool building. The approach is demonstrated on an example of a subset of UML activity diagrams. The benefits of the approach are also clearly indicated. These include a natural and intuitive definition of tools, a strict logic/presentation separation and the openness for extensions as well as for external applications.**

***Keywords* – Metamodel specialization, open architecture, submetamodel, supermetamodel, tool-building framework.**

I. Introduction

Nowadays we have a very wide range of different tools to choose from when searching for a tool for our specific needs. For example, if we need to create UML activity diagrams, we can use various UML tools like Enterprise Architect [1], IBM Rational Rose [2] or others. However, if we want to alter the editor a little bit (e.g., by adding some new elements or attributes), it is usually not easy to do. It can be done easily by members of the tool developers' team, but it can require a great knowledge of various specific details about the tool from the real tool user. In some cases we can get through by using stereotypes, but they can help only to a certain extent. It becomes even more difficult if we want to extend the tool functionality with some more specific features or to connect the tool to some external application. It usually requires a very deep understanding of the tool's architecture. If we turn to the world of domain-specific tools, the situation is a little better. Most of these tools usually offer quite a wide range of possibilities to alter the behavior of the tool. However, adding very specific features, which are not provided universally, again requires a deep understanding of various technical details.

The main goal of this paper is therefore to propose a conceptually different approach for developing graphical domain-specific tools. Instead of maintaining two distinct metamodels (domain and presentation) and synchronizing their instances using model transformations, we make one metamodel as a *submetamodel* of the other, thus liberating ourselves from all the synchronization issues. The approach will be demonstrated on several small examples based on UML activity diagrams [3]. The task would be to create an editor for those diagrams.

We want to look at the tool openness problem from two different points of view. The first kind of openness is the ability to alter the behavior of the tool. The tool builder must have a convenient and natural way of extending the tool functionality with a specific functionality. The other kind of openness is the ability to use the data from outer applications. There must be a way how we can access the tool repository from the outside and work with the data (change and process them, but, of course, only to some extent).

The rest of the paper is organized as follows. Some related work in the field is inspected in Section 2. In Section 3 we start with explaining our ideas about the specialization of metamodels which is a powerful facility itself. We base ourselves on the features of UML, and then we extend those by adding some useful possibilities found in OWL 2 [4]. Then we go on and show how this can be applied in a graphical domain-specific tool building. The benefits of the approach are clearly pointed out in Section 4.

II. The World of Tool Building

As said in the introduction, there are many domain-specific graphical tool-building platforms nowadays, like Punamu/Marama [5, 6], ViatraDSM [7] and Tiger [8]. Of course, we must mention MetaEdit+ by MetaCase company [9] and Microsoft DSL Tools [10] which are both very widely used industrial products. There are also solutions based on Eclipse [11] modeling framework, such as Eclipse GMF [12] and METAclipse [13]. Let us see how the job is being done in some of those platforms!

MetaEdit+ is known for its ability to be able to create a new graphical domain-specific tool in a very short time – in an hour or even half an hour. The tool will, of course, have only some basic functionality, but this functionality will be fully operating, which is a very useful facility of the platform. The tool creator can then spend more time and efforts polishing the tool and adding other non-standard features. The technical details of MetaCase solution are hidden, so their metamodels are not fully accessible from external applications. This is, of course, a very natural situation since this is an industrial tool, not fully academic. One flaw that, however, emerges from this concealment is that the tool creator must confine him/herself to the graphical presentations built in the platform. It is not possible for one to create his/her own presentation format

(e.g., some new kind of project tree depiction) from the domain data located in the repository.

In Microsoft DSL Tools, a tool creator is allowed to develop metamodels of the tool directly, which is a good feature. Also, the domain is clearly separated from the presentation, so the developer is very confident of what he/she is doing. A mapping between both metamodels (domain and presentation) can then be made by creating mapping lines in a visual editor. That, however, raises some issues. First of all, data redundancy is thus introduced which is never a good thing – partly the same information is stored in both metamodels. That makes it more difficult, for example, to navigate through instances if compared to the situation where classes of one metamodel would be subclasses of the other instead of being completely different ones. Besides, the mapping implementation forces both metamodels to be quite similar one to another in order to map them.

In Eclipse GMF, three models have to be specified first. They are a domain model, a diagram definition model and a diagram mapping model. The latter model establishes mappings between the first two. Then, from these three models (which together may be viewed as a platform-independent model, PIM, according to MDA principles [14]) are transformed to the Generator Model (which may be considered a platform-specific model, PSM). The generation parameters in the Generator Model may be adjusted, and then Java code is generated. The generated Java code can be manually edited to enhance the functionality of the generated DSL tool. This process requires a good knowledge of GMF. A toolsmith has to know how and when to use graphical wizards as well as when to generate certain artifacts. Since GMF focuses on GEF [15], it is not an easy task to add new graphical presentations for the developed tools.

Something very similar to the approach explained in further sections comes from the world of object-oriented programming. The notion of subclasses is also explored here as well as the possibility to redefine methods of a superclass in its subclasses. However, a big flaw concerning the tool building area is the inability to change the body of methods dynamically (although it can be simulated through function-typed attributes). Every method is attached to some class instead of being attached to some particular objects of that class. In tool building this feature turns out to be very critical in many situations where we want some objects of some class to behave differently in certain situations than other objects of the same class.

We must also mention here our own previous experience within the field of graphical domain-specific tool building. Two of our previous solutions (GrTP [16] and METAclipse [13]) were based on a special type diagram (in some sense – a tool definition metamodel) whose instances would be different graphical tools (a universal transformation interpreted those instances at run-time) [17]. Here all the necessary information about the tool was to be included in this metamodel which could then be linked to every presentation metamodel (and also domain metamodel) when needed.

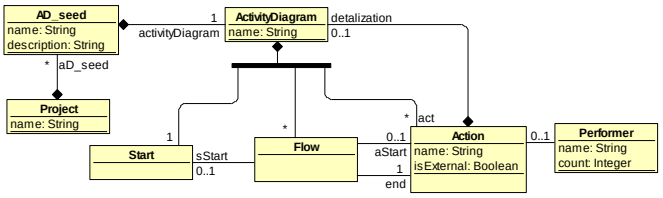

Fig. 1. The subset of UML activity diagrams used in further examples

This was a very elegant solution as far as we were developing quite simple tools whose behavior fit in well with universal behavioral patterns of the system. If some extra functionality was to be added, it could also be done by extending the universal tool-definition (metamodel-interpreting) transformation with manually written transformations. However, this extension was inconvenient in some situations since transformations were attached to classes of the tool definition metamodel instead of being attached to those classes having real run-time instances (so called "*this* pointers" for those transformations). This has been considered when developing the next-generation tool-building platform described in this paper.

III. The New Idea – Metamodel Specialization

In this section the ideas on the new method of building graphical domain-specific tools are explained. We start by discussing metamodel specialization and introducing the notion of a submetamodel. The UML generalization property is then supplemented by some extra facilities in order to achieve a very powerful metamodel specialization mechanism. This mechanism could perhaps be used in different areas of modeling and metamodeling. However, in this paper (in Section 3.3) we show how this mechanism can be applied in the field of graphical tool building. Instead of transforming instances of several metamodels, we use metamodel specialization, thus making the same instances belong to several metamodels simultaneously.

The approach will be demonstrated on an example of UML activity diagrams. Only a small subset of the real activity diagrams will be used (see Figure 1) on which all the different facilities can be shown.

A Basic Ideas

To understand the main idea correctly, let us first try to change the way we think about the UML generalization property. Traditionally, we perceive it as an assertion about a set of instances of some class being a subset of instances of some other class. However, we can also look at the generalization by changing it from an assertion into a command in a form "Make something as a special case of something else!". Thus now we can have two arbitrary classes, and we wish to make one of them a subclass of the other. So, at the moment of creating the generalization between those two classes, a command is executed throwing instances of the subclass into a superclass. This throw-in is made only virtually though (no data are being duplicated) – if traversing instances of the superclass, also instances of the subclass are visible.

Such kind of generalization is actually considered to be a mapping between two given metamodels. This kind of generalization can also be simulated using abstract classes. As for classes, the same applies to the generalization of other UML primitives – associations and attributes (this time the generalization is established between the sets of pairs of objects). Here and further we will use a word *association* when referring to what is actually an object property and a word *attribute* when referring to the data type property. Pairs of subassociation or subattribute are virtually thrown into the set of pairs of superassociation or superattribute. For both these properties this kind of generalization can be simulated using the concept of a derived union. Multiple inheritance is also allowed here.

Having such a notion of generalization in mind, we can now define a new activity – specialization of the metamodel. The metamodel specialization is a process in which we define a relation between two given metamodels by exploiting the new kind of generalization relation between elements of one metamodel (called the submetamodel) and the other (called the supermetamodel). This process is described in detail in the next subsection.

B Supplementing the UML Generalization

To make the metamodel specialization more powerful, we extend the metamodeling language by several new facilities that are not found in UML. We are actually looking towards OWL 2 and trying to borrow the most useful features from it. This subsection inspects all these add-ons, and therefore it can be considered the essence of this paper.

Specification of values for attributes of inherited class

Since creation of generalization is now considered to be a command, we can extend it with a "such that" part – "Make this class a subclass of that class such that the values for those attributes are as follows: …".

An example can be seen in Figure 2a. There is a class "Box" in the graph definition metamodel with such attributes as "shape", "bkgColor", "lineStyle" and, perhaps, others as well. Then we define its subclass "Action" (which comes from the UML Activity diagrams shown in Figure 1) saying that all instances of this class will be green (bkgColor VALUE clGreen) and will have a rectangular shape (shape VALUE shRectangle). In Figure 2b the same is depicted using a more compact syntax (similar to that of OWLGrEd, the OWL graphic notation editor [18]).

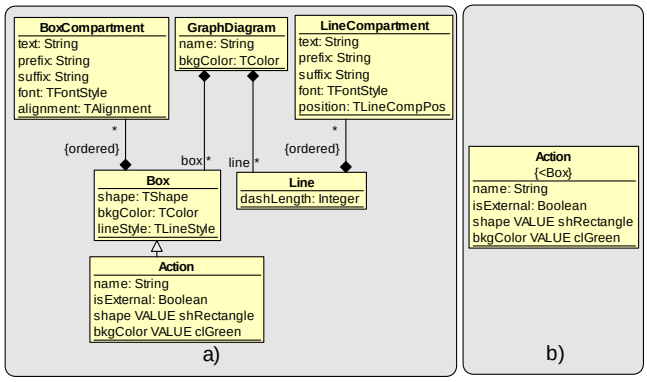

Fig. 2. Specifying values for attributes of inherited class "Action"; a) UML-style generalization notation; b) OWL-style generalization notation

Association concatenation

The second facility by which we extend the UML generalization is a concatenation of associations. This feature is also present in OWL 2. Having some association A in the supermetamodel, we can have several corresponding associations in the submetamodel, the sequence of which is then considered to make a virtual subassociation of association A. A good example of this feature is a definition of a project tree (see Figure 3). The activity diagram metamodel here is supplemented with two new virtual associations (depicted in points) whose physical representation is in each case a sequence of two real associations. Here, we use the OWL 2 Manchester syntax again to represent the concatenation as a small circle. For example, the virtual association between classes "Project" and "ActivityDiagram" is actually a concatenation of associations "aD_seed" and "activityDiagram" starting in the class "Project" (aD_seed o activityDiagram). This concatenation is then considered to be a subassociation of the association "root" starting in the class "Tree" which is a superclass of the class "Project" (therefore the syntax "{<root}"; from here on we will use the symbol "<" to denote the UML relations *subclassOf* (for classes) and *subsets* (for attributes and associations)). In such a way, we can skip technical objects in the navigation chains (e.g., the objects of the class "AD_seed" in the example of Figure 3). In OWLGrEd version the syntax would be a little different – an expression "{>aD_seed o activityDiagram}" would be attached to the superassociation "root" from the class "Tree".

In Figure 3 we can also see the attribute inheritance. Attribute "name" of the class "Project" is declared to be a subattribute of the attribute "tname" of the class "Tree". Also, the attribute "name" of the class "ActivityDiagram" is a subattribute. There is, of course, no request for the names of sub- and superattributes being different (e.g., "name" and "tname"). It is used here for a better understanding only.

Also, a predefined value for the attribute "picture" of the subclass "ActivityDiagram" is used in the example in Figure 3.

Synthesis and analysis of a superattribute

As we mentioned in the previous subsection, we use the OWLGrEd version of OWL 2 syntax to describe the fact that some attribute is a subattribute of some other attribute. That means those two attributes have some kind of connection that has to be taken into account when getting or setting the value of one of them. In the simplest case the subattribute can

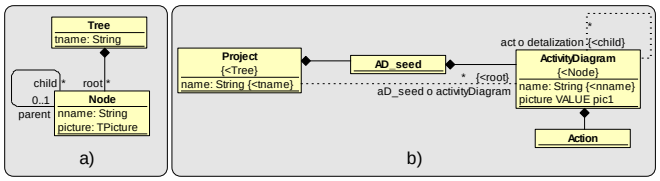

Fig. 3. Association concatenation; a) the project tree metamodel; b) association concatenation in the submetamodel

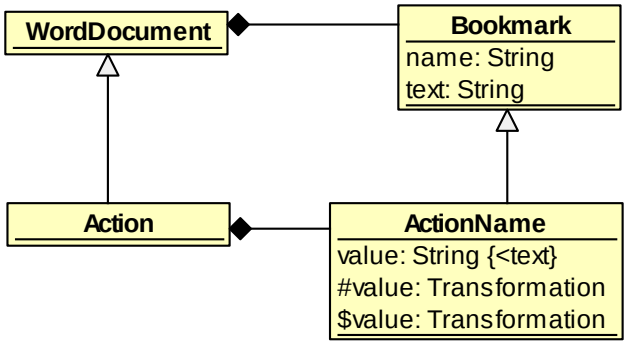

Fig. 4. Part of the UML activity diagram metamodel as a submetamodel of the Word document bookmark metamodel. The value of action name has a getter (#value) and a setter ($value) attributes whose names are names of transformation programs implementing them.

simply inherit the value from the superattribute. However, this connection cannot always be so straightforward and simple. In a general case we need to have a getter and a setter for every subattribute like in the classical object-oriented world. We introduce getters and setters as special attributes (written in a special syntax – see Figure 4) whose values are names of transformation programs implementing the particular getter or setter. In Figure 4 we can see one getter and one setter for the attribute "value" of the class "ActionName". This attribute is a subattribute of the attribute "text" of the class "Bookmark" which can be considered as the action name having some adornments.

When specifying subattributes we must depict them as classes in order to be able to attach getters and setters to them. In run-time, however, we can also refer to them as normal attributes. This feature is provided by the kernel we use in the Transformation-Driven Architecture (TDA, [19]). The kernel also gives the transformation writers a possibility to know nothing about the getters and setters at all. We can use some attribute as usually, and the kernel will call the getter or setter for that attribute if needed to retrieve its value for us, or to set it. For example, in Figure 4 we can simply write something like "s = obj.getAttrValue(value)" in our transformation program to get the value of the attribute "value" of the class "ActionName". The TDA kernel now intercepts that request and scans the class "ActionName" for the getter of the attribute "value". If it finds the getter, it is called and the return value is calculated (perhaps from the value of the attribute "name" of the class "Bookmark"). Similarly in the other way – we can write something like "obj.setAttrValue('value','myValue')" to set the name of the attribute. Now the kernel scans for a setter which is then called to set also the value of the attribute "name" of the class "Bookmark" if needed.

The overall schema of the Transformation-Driven Architecture is seen in Figure 5. It is, however, out of the scope of this paper to go into a greater detail about the TDA here. A more detailed description of the architecture can be found in [19].

Getters and setters are optional. If we turn to some attribute not having a getter and want to get its value, the kernel recognizes the situation and returns the direct value of its

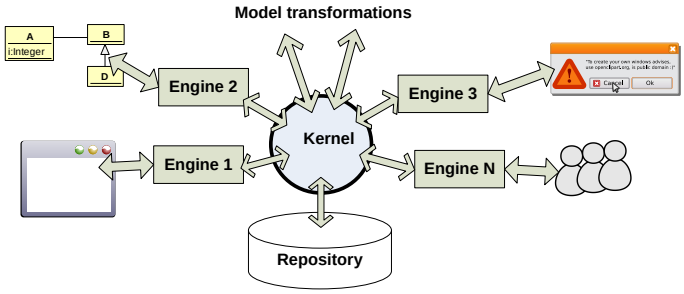

Fig. 5. The Transformation-Driven Architecture

superattribute or the value of the same attribute if it is not inherited.

C Application of Our Ideas in the Field of Tool Building

The metamodel specialization ideas explained in the previous subsection can be used in several areas. The intention of this paper is to apply these ideas to the field of domain-specific graphical tool building. It is a common practice in tool-building platforms to maintain two metamodels – the domain metamodel representing the particular domain (its abstract syntax) and the presentation metamodel representing the way the user sees the domain (its concrete syntax). Then there is usually some kind of mapping between those two metamodels, and the data are always synchronized somehow. This schema requires a lot of work to be done in case a new tool is needed. The domain metamodel needs to be designed, then the mappings must be constructed, and the data synchronization algorithm must be provided. Regardless of whether some of these tasks can be automated in most cases or not, the whole structure of the tool turns out to be quite complicated.

The idea here is to specialize the presentation metamodel so far till we get the desired domain metamodel. So the domain metamodel would be a submetamodel of the presentation metamodel which sets us free from the need to have some extra mappings and/or sophisticated data synchronization facilities. We can see here a similarity to a sculptor sitting and watching a stone block until he finally sees a sculpture in it (and then carves it out). In the same manner a tool developer looks at the particular presentation metamodel and sees how it can be specialized so to get the particular domain metamodel out of it. Since we allow also multiple inheritance of classes, associations and attributes, multiple presentation metamodels can exist simultaneously, and multiple specializations can thus lead to the same domain metamodel.

Some examples of such a specialization can be seen in the previous figures. In Figure 2 we can see a presentation metamodel being a graph diagram visualization metamodel. This presentation metamodel can now be specialized to get the metamodel for our specific domain – UML activity diagrams. So, for instance, the class "Action" would be made as a subclass of class "Box" as shown in Figure 2. Other classes of the activity diagram metamodel would also be made subclasses of some of these five classes of the graph diagram metamodel. Of course, other classes, not being subclasses of

any presentation metamodel class, are as well allowed in the domain metamodel.

In Figure 3 the presentation metamodel is a project tree metamodel consisting of two classes – "Tree" and "Node" (see Figure 3a). This metamodel is now again specialized to get the same domain metamodel of UML activity diagrams. Since multiple inheritance is allowed, both presentation metamodels – the graph diagram visualization metamodel and the tree metamodel – can exist simultaneously. Each presentation engine (in terms of Transformation-Driven Architecture [19]) "sees" only its own presentation metamodel and is able to interpret it as needed. Other parts of the platform responsible for the semantics of data (e.g., transformations) can operate with the domain metamodel and be not much concerned about the presentations or any mappings.

A good tool building platform must be intuitive in its usage allowing defining tools in a natural way. This is now provided through the new perception of UML generalization relation. The tool is to be built by just creating the domain metamodel as a specialization of some presentation metamodels. There is no need to have any mapping between them. However, it is clear that only quite simple tools can be made this way. In order to meet the two openness requirements mentioned in the introduction, we must allow the tool developer to extend the tool functionality by some specific facilities by providing a way to append his/her own model transformation programs. This is done by creating so called extension points and adding them to classes of the domain metamodel.

Technically, each extension point is an attribute whose value is the name of the transformation program to be called at a specific moment. At this point, such type of attributes is nothing new – getters and setters mentioned in Section 3.2.3 were the same type of attributes. This type of attributes could resemble the methods of classes in the object-oriented world – the transformation programs (whose names are values of those attributes) can also use something like "this" pointer pointing to the object of the given class for which the transformation is called. However, unlike methods, we allow a dynamic change of those transformation programs (which is being done by changing the value of such attribute). In OOP dynamic change of a method body is not possible, but it can be simulated using function-typed attributes.

Some examples of extension points can be seen in Figure 6. For instance, in the class "Action", there is an extension point "afterCreate" which is called immediately after a new action is created. So, here one can insert his/her own transformation program and change the default behavior of the action-creating process. If we do not want to change anything in this process, we can also set the value of the attribute "afterCreate" to an empty string which means no transformation will be called at that specific moment.

Having this powerful extension mechanism, it must be clear that we can extend the tool with all the extra functionality needed by simply coding it in a model transformation language.

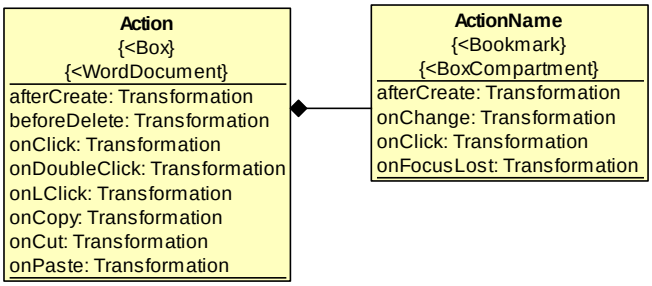

Fig. 6. Some of possible extension points attached to domain classes. Here we can specify if and which transformation programs need to be called at specific moments.

The third requirement for the platform – the accessibility from external applications – is achieved by fully providing the domain metamodel and some API functions for working with it. Again, when some external application makes an API function call, it is intercepted by the TDA kernel as explained before which can then decide what to do (either to pass the call further to the repository or to perform some extra actions as in the case with attribute getters and setters). Since the domain metamodel is a submetamodel of every presentation metamodel, external applications do not need to know anything about all those presentations or any mappings – all the information of the domain is as if automatically synchronized with the presentations.

IV. Benefits of Our Approach

Now when we have described our approach in domain-specific graphical tool building, a question may arise what are the benefits of this approach over other tool definition methods. Some benefits were already mentioned in the previous sections. In this section the main benefits are clearly summarized.

A Naturalness of the Tool Building Process

This feature of the tool building process was already mentioned in Section 3.3. The only thing we have to do when creating a new tool is to develop the domain metamodel as a specialization of our existing presentation metamodels. We do not set any restriction to the domain metamodel in terms of what must be depicted as classes and what – as lines. For instance, if we want some domain association to be depicted as a box in the presentation (or vice versa), it can also be done by combining the facilities described in Section 3.2. We can see such an example in Figure 7.

In Figure 7a a situation is depicted when we want to hide some technical domain class T from the presentation. The association concatenation does the job here and, when reading the presentation association pa, the domain association concatenation a o b is actually read.

In Figure 7b our intention is to depict the domain association as a box in the presentation. A technical class T is introduced, and the given association is made as a concatenation of associations a and b. Now, if we want to create a new box in the presentation concerning this association, a new object of class T will also be created. Here,

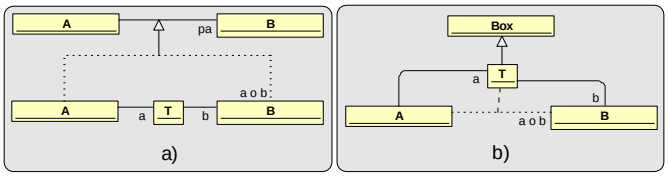

Fig. 7. Different usages of association concatenation; a) avoiding some domain technical class; b) introducing some domain technical class as an association class

the concatenated association is dotted as before, and the dashed line denotes the UML syntax for creating the association class.

The tool building process is not always so natural in other tool building platforms. Therefore more constraints to the domain metamodel have to be considered. For instance, in MetaEdit+ the domain classes can only be depicted as boxes in the presentation, and the domain associations can only be depicted as lines. In Microsoft DSL, however, the visual representation of a domain association can be specified.

B Strictly Separated Logic and Presentation

The logic and the presentation of the tool in the creation are strictly separated in this approach. Be it the graph visualization, the tree, the dialog windows or some other presentation, its engine is operating only with its metamodel. It does not have to deal with other presentations or the logic part – submetamodels of its metamodel (see Figure 8).

On the other hand, model transformations interpreting the domain metamodel and thus incorporating the logic part into the tool does not have to deal with all the presentations built upon that metamodel. Meanwhile, external applications can work only with the domain. Such a strict division makes it easier for more advanced users to write their own transformations and add them as extension points. It is also less error-prone. The same applies to external applications – they can perform well knowing only the domain part of the tool.

Separation of domain and presentation metamodels is a common practice in many tool building platforms. However, it is usually done at the physical level making two real metamodels and then trying to map them somehow. As a result data redundancy can occur, as well as all the problems connected with the mapping and data synchronization issues.

C Openness for External Applications

An interface in the Transformation-Driven Architecture [19] is a pair of an engine and a metamodel that the engine understands.

As mentioned before there can be interfaces like graph visualization interface, tree interface and dialog window interface. These and other interfaces are built-in interfaces in TDA. However, advanced users can also add their own interfaces easily. To add an interface (e.g., an external application), we first have to develop the interface metamodel (called the presentation metamodel in the previous sections) and the interface engine. Then, we have to add this couple to the platform by means of TDA. Now new domain metamodels can be made as specialization of this interface metamodel.

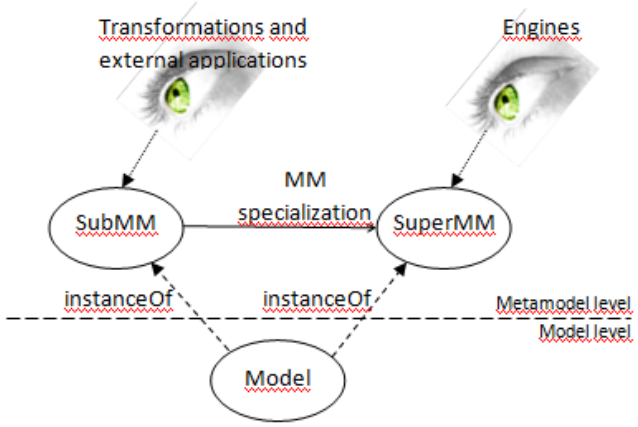

Fig. 8. The domain-presentation division allowing each part of the platform to operate only with metamodels relevant to them

Besides, they can be specializations of some other interface metamodels as well. There is even no need to alter the universal interpreter interpreting user events at run-time – all the semantics of this new interface metamodel can be put into extension points of the domain metamodel. The external application can then work with the domain metamodel and its models.

Figure 9 illustrates this situation. The approach does not depend on how many different presentations there are in the platform. One can also add new presentations easily. In other platforms (like in all major ones – MetaEdit+, Microsoft DSL Tools and Eclipse GMF) there are some built-in presentations. In order to add some new presentation we would have to predefine the C# or Java code to make the presentation capable to map with the domain. This can be quite difficult in many situations.

D Openness to Extensions

Extension points are technically coded as attributes whose values are names of transformation programs to be called at specific moments. Regardless of quite intuitive extension point names, these calling times are precisely specified so that every transformation writer knows exactly when his/her transformation will be executed at run-time. The transformation writer knows the class whose objects will be pointed to by "this" pointer – that will be the same class for which the extension point is defined.

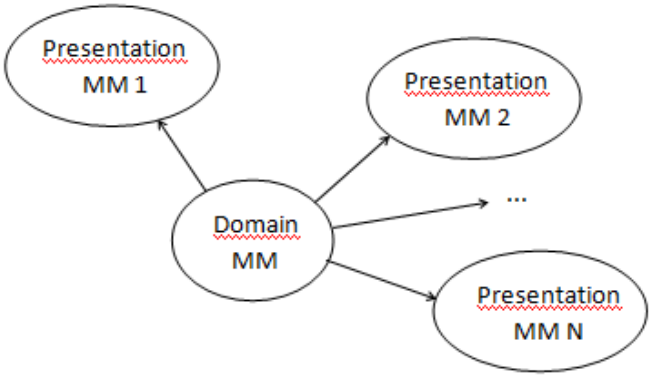

Fig. 9. Domain metamodel as a specialization of presentation metamodels. When adding a new presentation, only the specialization relations must be specified

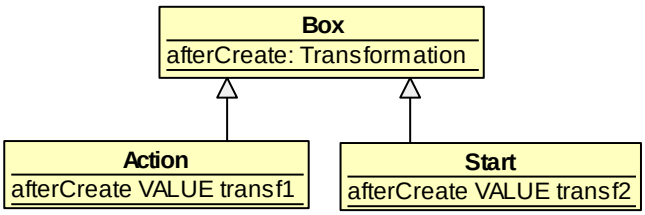

Fig. 10. Introducing extension points into a superclass and specifying them in subclasses

For example, an extension point "afterCreate" can be defined for the presentation class "Box" (see Figure 10). In submetamodels of that presentation metamodel, the value of this attribute can now be fixed differently for every subclass of "Box" in the specialization process. Here different transformations will be called after creating objects of the class "Action" and objects of the class "Start". This facility is almost the same as virtual methods in object-oriented programming.

This part of transformation writing is made very clear in the presented approach. In our previous approaches (GrTP [16] and GRAF [20]) the chaos with the extension point attachments was the biggest flaw – they were attached to other classes than those whose objects they were called on.

V. Conclusions

In this paper a novel approach to graphical domain-specific tool building has been presented. To develop a new graphical tool, we must specify its domain metamodel and provide specialization relations between its elements and the elements of one or more existing presentation metamodels. In this specialization process several facilities can be used to make the tool more powerful – the specification of values for attributes in inherited classes, the association concatenation and the superattribute synthesizing and analyzing features. Additionally, we allow extending the tool with manually-written function calls in the form of extension points attached to domain classes.

Having such a specialization facility, we obtain several benefits which have been described in detail in this paper. The tool creation process is very natural, the presentation and the domain are strictly separated without introducing various synchronization problems and data redundancy. So the platform is open for external applications as well as for manual extensions. We believe that the metamodel specialization is quite a powerful feature itself to have also several other use-cases outside the field of domain-specific tool building which are not covered in this paper.

A very important feature in domain-specific tools for business usage is its openness for external applications which was mentioned before. A typical situation is to build a domain-specific tool for editing business processes (e.g., a tool for project assessment processes that we have built as a case study within the framework of our study [21]). We can then create various business process diagrams with the tool. Such a diagram is usually sufficient for an employee whose task is to understand the business process. However, for the person submitting the process (in case of project assessment diagram tool [21]) the plain diagram could not be enough – one could wish to see how far the project has gone in the assessment process. Therefore the platform can be easily extended (and actually is extended) with the possibility to export data to the Scalable Vector Graphics format (SVG [22]). This is coded in the form of a domain metamodel which has to be understood by the outer application (in this case – the base information system developer). The SVG data can now be visualized, for example, in a web browser, and a path of the current project in the assessment process can be drawn. This can be considered a good example of the openness feature possessed by the platform.

Acknowledgments

This work has been supported by the European Social Fund within the framework of the project "Support for Doctoral Studies at University of Latvia".

References

1. Enterprise Architect, http://www.sparxsystems.com.au
2. IBM Rational Rose, http://www-01.ibm.com/software/awdtools/developer/rose
3. OMG modeling specifications, UML 2.1.1 Superstructure and Infrastructure, http://www.omg.org/docs/formal/07-02-05.pdf
4. Web Ontology Language (OWL), http://www.w3.org/2004/OWL
5. Nianping Zhu, John Grundy, and John Hosking. Pounamu: a meta-tool for multi-view visual language environment construction. 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC'04), 2004, pp. 254-256.
6. John Grundy, John Hosking, Jun Huh, Karen Na-Liu Li. Marama: an Eclipse Meta-toolset for Generating Multi-view Environments. ICSE'08, Leipzig, Germany, 2008.
7. I. Rath, D. Varro. Challenges for advanced domain-specific modeling frameworks. Proc. of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France.
8. C. Ermel, K. Ehrig, G. Taentzer, E. Weiss. Object Oriented and Rule-based Design of Visual Languages using Tiger. Proceedings of GraBaTs'06, 2006, pp. 12.
9. MetaEdit+ Workbench User's Guide, Version 4.5, http://www.metacase.com/support/45/manuals/mwb/Mw.html, 2008.
10. S. Cook, G. Jones, S. Kent, A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.
11. Eclipse. http://www.eclipse.org
12. Graphical Modeling Framework (GMF, Eclipse Modeling subproject), http://www.eclipse.org/gmf
13. 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.
14. MDA Guide Version 1.0.1. OMG, http://www.omg.org/docs/omg/03-06-01.pdf
15. Graphical Editing Framework (GEF, Eclipse Modeling subproject), http://www.eclipse.org/gef
16. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis. GrTP: Transformation Based Graphical Tool Building Platform. MODELS 2007, Workshop on Model Driven Development of Advanced User Interfaces, 2007.
17. J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. MDE-based Graphical Tool Building Framework. Scientific Papers, University of Latvia, "Computer Science and Information Technologies", Vol. 756, 2010, pp. 121-138.
18. J. Bārzdiņš, G. Bārzdiņš, K. Čerāns, R. Liepiņš, A. Sproģis. OWLGrEd: a UML Style Graphical Notation and Editor for OWL 2. OWLED 2010,

OWL: Experiences and Directions, Seventh International Workshop, 2010.
19. J. Barzdins, S. Kozlovics, E. Rencis. The Transformation-Driven Architecture. Proceedings of DSM'08 Workshop of OOPSLA 2008, Nashville, USA, 2008, pp. 60-63.
20. A. Sproģis, R. Liepiņš, J. Bārzdiņš, K. Čerāns, S. Kozlovičs, L. Lāce, E. Rencis, A. Zariņš. GRAF: a Graphical Tool Building Framework. Proceedings of the Tools and Consultancy Track. European Conference on Model-Driven Architecture Foundations and Applications, Paris, France, 2010, pp. 18-21.
21. J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. Domain Specific Languages for Business Process Management: a Case Study. Proceedings of DSM'09 Workshop of OOPSLA 2009, Orlando, Florida, USA, 2009, pp. 34 – 40.
22. Scalable Vector Graphics (SVG) 1.1, http://www.w3.org/TR/SVG

Edgars Rencis, Mg. Sc. Comp., has received the Master's Degree in Computer Science at the University of Latvia, 2007. In 2010, he finished the doctoral program and is yet to defend his PhD thesis. His major field of study is the development of graphical tool-building platforms. He is now working at the Institute of Mathematics and Computer Science, University of Latvia as a RESEARCHER. Before that he worked as a PROGRAMMING ENGINEER at the same institute. Concurrently he is working as a LECTURER at the University of Latvia and at Vidzeme University. His current and previous research interests include tool-building platforms and model transformation languages.

Janis Barzdins is full PROFESSOR in Computer Science at the University of Latvia. From 1997 to 2006, he was also DIRECTOR of the Institute of Mathematics and Computer Science, University of Latvia. He received his Doctor of Science degree (Mathematics) in 1976 from the Institute of Mathematics (Novosibirsk), Academy of Sciences of the USSR. He has also worked in industry on developing system modeling tools. His current research interests include system modeling languages and tools, as well as system engineering methods based on metamodeling and model transformations. Since 1992, he is also full member of the Latvian Academy of Sciences.

Sergejs Kozlovics, Mg. Sc. Comp., has received the Master's Degree in Computer Science at the University of Latvia, 2008. He is now about to finish the doctoral program at the same university. His major field of study is the development of graphical tool-building platforms and their architecture. He is now working at the Institute of Mathematics and Computer Science, University of Latvia as a RESEARCHER. Before that he worked as a PROGRAMMING ENGINEER and ASSISTENT at the same institute. Concurrently he is working as a LECTURER at the University of Latvia.