SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 121–138 P.

An MDE-Based Graphical Tool Building Framework

Janis Barzdins, Karlis Cerans, Sergejs Kozlovics, Lelde Lace,
Renars Liepins, Edgars Rencis, Arturs Sprogis, Andris Zarins
Institute of Mathematics and Computer Science
University of Latvia, Raina bulv. 29, Riga, LV-1459, Latvia
{Janis.Barzdins, Karlis.Cerans, Sergejs.Kozlovics, Lelde.Lace,
Renars.Liepins, Edgars.Rencis, Arturs.Sprogis, Andris.Zarins}@lumii.lv

In this paper, an MDE-based approach to tool building is described. It is based on a core tool definition metamodel and an interpreter of this metamodel. Besides, an extension of the core metamodel is proposed, allowing for tool-specific model transformations to enrich the behavior of the universal interpreter. As a result, a novel wide-profile tool building platform is obtained. The visualization component of the platform is based on an original high-performance graphical diagram presentation engine which embodies advanced graph drawing algorithms.

Keywords: tool definition metamodel, tool building platform, model transformations.

1 Introduction

In this paper, we present an MDE-based interpretive approach to domain-specific tool (DST) building on the basis of a simple yet flexible and powerful tool definition metamodel (TDMM) that fully specifies a DST as its instance, and the interpreting engine of this metamodel.

The idea of providing explicit metamodeling foundations for the meta-tools themselves has not been central to many powerful developments in DST building area, including MetaEdit+ [1, 2], Pounamu/Marama [3, 4], ViatraDSM [5], Tiger [6], and METAclipse [7]. These tool-building frameworks generally offer some configuration facilities that allow us to define a DST in a user-friendly way (for instance, Pounamu [3] offers a shape designer, metamodel designer, event handler designer and view designer, MetaEdit+ [1] offers Object, Relationship, Role, Port, Graph and Property tools). The result of the configuration process, however, is typically stored in some format that is not revealed to the tool user and that is later compiled or interpreted to obtain a DST.

Our approach advocates opening the tool runtime structures to the end user in the form of a simple metamodel that specifies the DST as its instance. The organization of the DST definition and runtime structures in the form of a simple metamodel, in addition to its theoretical appeal (applying MDE principles to meta-tools supporting MDE-based development themselves), allows for possibilities of basic tool behavior extension by (high-level) model transformations that we ascribe to certain well-defined extension points in the tool definition metamodel and that are handled by the tool metamodel interpreting engine. These transformations can be used for, e.g., domain

model synchronization, constraints, dynamic content in the tool, advanced presentation behavior, as well as integration with other data engines.

Some further benefits of the metamodel-based tool data structure include model migration possibilities among tool and meta-tool versions just by model transformations, as well as easy external access to model repository.

The idea of DST definition by a metamodel is already successfully implemented, for instance, in the Eclipse GMF framework (GMF) [8] (Microsoft DSL Tools (DSLT) [9] also follow a related approach). In GMF, a tool definition consists of instances that correspond to domain, graphical definition, tooling and mapping metamodels, and the tool itself is obtained by compiling these instances into a Java code. The main difference of our approach from that of GMF or DSLT is that we aim for greater flexibility of MDE-level constructs in tool definition by following the tool model interpretation approach (instead of compilation into JAVA or C#). On the basis of this approach, we are able to offer the possibility to extend tool behavior by means of model transformations that can be attached to certain well-defined points in the tool definition metamodel. In GMF or DSLT, the tool behavior extension is possible by adding code to the JAVA or C# classes generated for the tool by the framework. This task may be feasible; however, it requires rather profound expertise in the internal program-level structure of classes and methods generated by the corresponding framework. Our approach provides an alternative to GMF and DSLT by allowing us to create the extensions in model-level rather than program-level terms.

We structure the presentation of the TDMM into core and extended versions, where the core metamodel allows for basic tool behavior description and the extended TDMM allows for model transformation incorporation. In Core TDMM, we focus on tools defined directly in terms of their graphical presentation (there are applications where this is sufficient). The handling of domain model, if that were necessary, is delegated to model transformations (allowed by the extended TDMM) that can perform the task (see, e.g., [7] for comparison of static mapping and model transformation approaches in modeling tools).

The TDMM is defined to contain both the tool definition and tool runtime instances at the same metamodeling abstraction level. This is achieved using a structure that resembles an adaptive object-model [10] element type pattern. A theoretical note: this design allows for easy implementation of dynamic tool model reconfiguration in parallel with particular model creation by the tool, as advocated, for instance, in [3] (in practice the modeling power of the platform is restricted in its “end-user” versions and user model migration between tool versions (as well as between platform versions) is achieved by model transformations).

The TDMM is also defined as an extension of a more general graph diagramming metamodel (GDMM) [11], and its implementation is provided by universal (platform-level) model transformations associated with the events (instances of Event class) defined in GDMM that interpret the specific TDMM instance. To make our presentation complete, we also review GDMM in this paper. An earlier authors’ work with much more limited tool definition possibilities and without separating GDMM from the tool definition metamodel has been reported in [12].

The rest of this paper is organized as follows. Section 2 describes the graph diagramming metamodel and engine (GDE), explaining what is used as the basis for

the tool-building platform. The communication mechanism between the graph diagram presentation engine and the "business logic engine" that is behind any specific graph diagramming tool and is typically implemented by model transformations (on the basis of GDMM) is also outlined here.

Section 3 presents the core tool definition metamodel, including its full abstract syntax and explanation of its semantics. This metamodel is general enough to allow for a definition of a broad class of DST, including the EMOF [13] class diagram and UML 2.0 activity diagram editors; yet it is simple enough for its abstract syntax, together with the relevant parts of GDMM, to be presented on a single page. We also outline principles of implementing the core tool building platform on the basis of GDE.

In Section 4, we extend the core tool definition metamodel to allow for MDE-based extension mechanism to the platform. It is a widely accepted fact that extensions are among the most complicated problems every meta-tool faces. The extension mechanism we propose is not hidden in the depths of implementation; instead, it is elevated to the level of the metamodel. The core tool definition metamodel together with extensions is sufficient to build efficiently, e.g., a full UML 2.0 class diagram editor with full support of attributes, stereotypes and tagged values, as well as other DST editors of comparable complexity. The high-level extensibility mechanism based on model transformations allows us to achieve such tool features that are beyond the scope of usual DSTs.

2 The Graph Diagramming Metamodel and Engine

The tool definition metamodel together with its interpreter – the tool building platform – are based on basic presentation services whose interface is described by metamodels. One of the most important such services is that of graph diagramming. It is defined by means of a graph diagramming metamodel (GDMM) and implemented by a graph diagramming engine (GDE). Another service for which we also have a metamodel and a corresponding engine is that of property editors. The property editor metamodel and engine are used in our implementation of the tool building platform; however, they are not of primary importance in explaining its semantics. Therefore, they are not considered in detail here.

The aim of GDMM is to describe the graph diagramming functionality that can be offered by GDE and that is common to a wide range of graphical diagramming tasks that may go beyond any particular DST, or even the task of DST building in general. Since providing appropriate abstractions in GDMM can considerably ease the tool definition process on the basis of GDE, the emphasis in the design of GDMM has been on properly separating our concerns into "purely graphical" tasks that are to be handled by the GDE itself, and tasks involving "logic" of the tools using GDE.

GDMM (Fig. 1) is built around the classes for visual elements of the presentation, namely *GraphDiagram*, *Element*, *Box*, *Line*, and *Port* together with *Compartment* corresponding to text fields placed in boxes and attached to lines and ports (note that the start and end of lines can be attached to any elements, not just boxes). Instances of these classes are diagrams and elements created by the user. Every element, compartment and graph diagram has its style (see classes *ElemStyle*, *CompartStyle* and *GraphDiagramStyle*). The metamodel allows for every element to specify its default

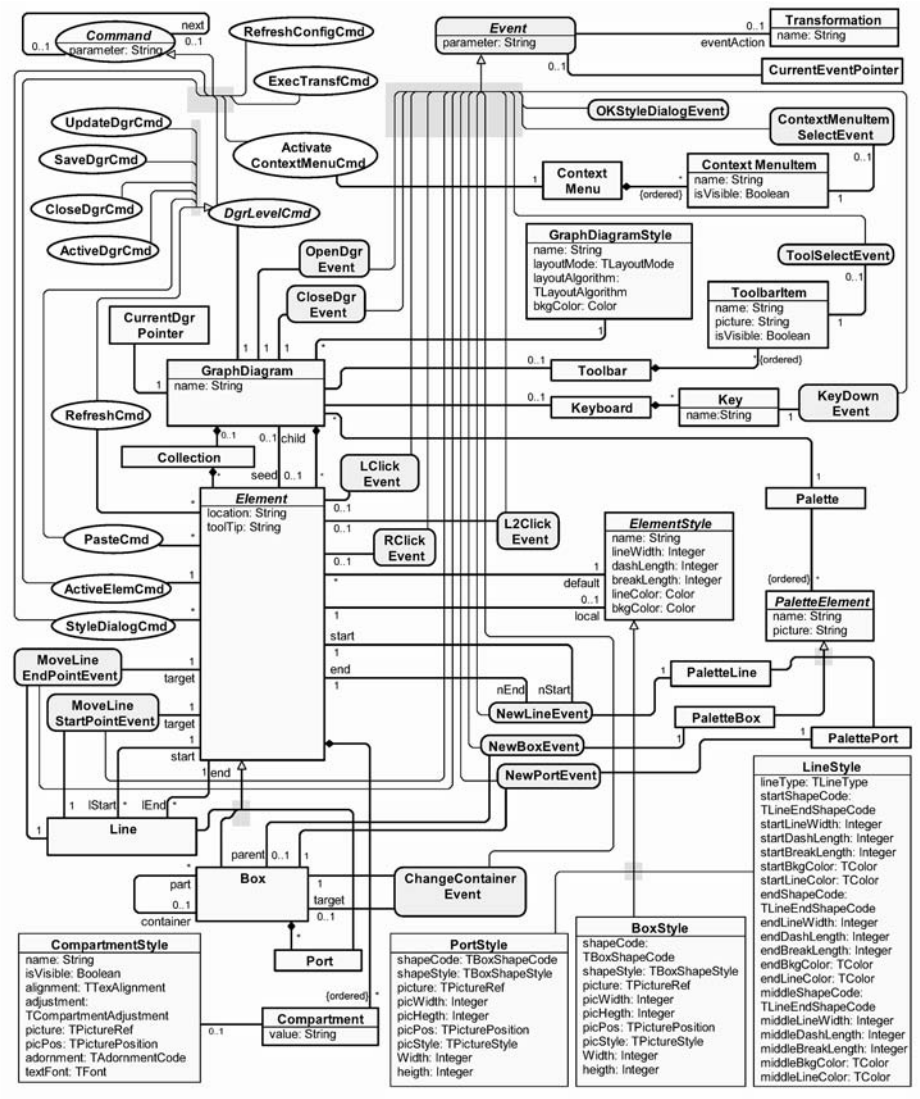

Fig. 1. The graph diagramming metamodel

style and local style (the diagramming engine uses the local style if it is defined; otherwise the default style is used). The *Collection* class contains a single item that is linked to currently active (selected) elements in the diagram. The *seed*/*child* link between *Element* and *GraphDiagram* permits specifying an element to be a seed for a diagram (typically, not the diagram the element is in), thus providing means for building diagram hierarchies.

Besides the classes of visual elements, GDMM also contains classes describing the tool's environment (*Palette*, *Toolbar* and *Keyboard* classes with corresponding elements). Instances of these classes are typically created at the tool creation time and

do not change during the work with a tool. A context menu (*ContextMenu* class) can also be specified to be opened in response to the tool's request.

There is an *Event* class in GDMM whose singleton subclasses correspond to the actions the user may perform on a particular diagram (the event classes are represented as rounded rectangles), and that are understood by GDE. Upon observing a current event, GDE invokes the event's *eventAction* transformation responsible for particular tool's "business logic" in response to this event. The *Command* class describes the requests (commands) that the tool transformations can issue for GDE. There may be several commands issued by a single tool transformation. Command classes are denoted as ellipses in Fig. 1.

For instance, the creation of a new box in a graph diagram starts by the user clicking the tool-triggering GDE to set *CurrentEventPointer* to the only instance of *NewBoxEvent* (the *parent* link from the event is set if the new box is to be created inside another box). The event's transformation then may, for instance, create a new element of the *Box* class (or it may do some extra/other action depending on the tool's specific logic). Then it creates an instance of *UpdateDgrCmd* and transfers the control back to GDE that processes the command by updating the diagram so that the newly-created box becomes visible.

The semantics of some further *Command* subclasses is explained as follows. The *ActiveDgrCmd* sets the editor's focus on the particular diagram, *RefreshCmd* refreshes the specified elements in the diagram, *PasteCmd* computes coordinates of elements pasted into the diagram model, *RefreshConfigCmd* rebuilds toolbars and palettes, *ActivateContextMenuCmd* opens a context menu (depending on the collection of elements pointed to by the *Collection* element), *StyleDialogCmd* opens the style dialogue of elements, *ExecTransfCmd* is used for calling back transformations. The other commands and events should be mostly self-explanatory.

Although most of user activities in a tool trigger setting of the current event and invoke some transformation, there are actions that are performed solely by GDE (e.g., undo/redo, zoom, export to HTML, print diagrams, etc). The toolbar items responsible for these actions do not have associated *ToolSelectEvent*s to be triggered when the user selects the toolbar item. The context menu item that is handled directly by GDE is "Symbol Style". GDE is also responsible for handling element coordinates (the coordinates can be abstracted away while writing tool defining transformations).

The implementation of GDE has been a considerable programming task of several person years. The relatively simple diagram structure has allowed us to implement advanced graph drawing capabilities [14, 15] in GDE, which support diagram initial layout application as well as serve the interactive diagram editing process. The definition of GDE interface in the form of GDMM allows for reuse of its graph diagramming capabilities in various MDE-related tasks, among them, meta-tool creation. The architecture of GDE is described in more detail in [11, 16].

3 The Tool Definition Metamodel: the Core

In this section, we describe the syntax and semantics of the core tool definition metamodel (Core TDMM) that can have (simple) modeling tools as its instances. The aim of Core TDMM is to provide basic means for DST definition on the level of graphical

presentation. There is a wide range of applications where the graphical presentation view on the modeled system is sufficient since this is the view of the system directly perceived by the user. The other views on the system if necessary can be obtained by model transformations that work either offline, performing export and import tasks, or synchronously, using tool behavior extension points, as described in Section 4.

Core TDMM (Fig. 2) is built around the concepts of *GraphDiagramType*, *ElementType* and *CompartmentType*, providing type (or pattern) information for graph diagrams, elements and compartments that are specified in the graph diagramming metamodel (GDMM) and that may appear in the particular tool's visual editor. Therefore, Core TDMM is described as an extension of GDMM. Fig. 2 describes the classes of Core TDMM as well as a selection of relevant classes of GDMM (in two grey rectangles).

The containment hierarchy *Tool* → *GraphDiagramType* → *ElementType* → *CompartmentType* (via *base* link) forms the backbone of TDMM. Every tool can serve several graph diagram types (one of these being the *first* diagram type in the tool). Every graph diagram type contains several element types (instances of *ElementType*), each of them being either a box type (e.g., an Action in the activity diagram), a line type (e.g., a Flow), or a port type (e.g., a Pin). Every element type has an ordered collection of *CompartmentType* instances attached via its *base* link. These instances form the list of types of compartments of the diagram elements of the particular element type.

We notice the resemblance of relations between graph diagram and graph diagram type, element and element type, and compartment and compartment type to adaptive object model [10] patterns.

The element type specification (*ElementType* class and its subclasses) allows to describe inclusion possibility between boxes of different types (*partType*/*containerType* relation), attachments of ports to boxes, the box type multiplicity constraints (e.g. 0..1 boxes of certain type in a diagram), as well as line type connectivity rules (the element type pairs for which connection by a line of a certain type is possible are specified by *LineSubtype* class instances).

The *CompartmentType* class is divided into subclasses according to the multiplicity of the type's compartments in the elements as well as the possibilities to work with them in the property editor. Table 1 summarizes these subclasses.

Table 1

Compartment type subclasses

FieldType	Single input field.
MultiLine FieldType	Multi-line input field, with each line corresponding to a compartment. The empty field corresponds to no compartments of this type in the element.
LabelType	Non-editable label. Used, for instance, in the property editor to show element names.
CheckBox Type	Check box. The attribute *displayValue* defines the value shown in the diagram when the user has selected the corresponding value. For instance, in a class diagram, when an attribute is derived (the corresponding check box is selected, activating a *CheckBoxItem* with value **true**), it should be displayed in diagram as "/".
ComboBox Type	Combo box. The user can choose among certain values predefined as *ComboChoiceItems*.

ComplexType	Compartment consisting of several sub-compartments. It can be entered either directly (e.g., as a string "attr:Integer=5"), or in a separate window, where values for sub-compartments (e.g., "attr" for the name, "Integer" for the type, and "5" for the default value) can be entered. The compartment's value is obtained by concatenating the values of its sub-compartments supplemented with the corresponding prefixes (like ":" and "=") and suffixes. See *displayPrefix* and *displaySuffix* attributes in *CompartmentType*. Note that to support the compartment hierarchy persistence beyond the element's editing time as well, we have introduced a *subCompartment* link from GDMM's *Compartment* class to itself in TDMM.
MultiLine ComplexType	Multi-line input field, where each line is a compartment of *ComplexType*.

In TDMM, there are diagram, element and compartment styles from GDMM connected to diagram, element and compartment types, determining how the diagrams, elements and compartments of the corresponding types are visualized (see Fig. 1 for style attributes). Apart from specifying the default style for diagram, element and compartment types, TDMM allows for the so-called optional styles of element and compartment types that can be triggered to become effective for a particular element/compartment, selecting a certain choice item in (possibly another) compartment of *CheckBoxType* or *ComboBoxType* (the links *elemStyleByItem* or *compartStyleByItem* from the *ChoiceItem* to the particular style instance are used). A classical application of this feature is putting or canceling the formatting of the class name compartment in italics depending on the value of class attribute isAbstract; however, this feature is much more useful.

Another form of dynamics supported by Core TDMM is adding compartments of new types to the elements depending on some compartment's value selected in a combo-box. This dynamics is implemented by defining instances of *DynamicCompartTypes* class as well as setting their dependencies from their triggering combo-box choice items, the position where the new compartments go, as well as the list of new compartment types themselves. This dynamics may be useful, for instance, in implementation of tagged values associated with stereotypes.

In TDMM we extend the GDMM *Compartment* class by the *inputValue* attribute, so that every compartment has both *inputValue* and *value* attributes. The *value* attribute to be displayed in the diagram is obtained from *inputValue* by prefixing it with compartment type's *displayPrefix* and suffixing it with *displaySuffix* (an example of this construction is putting double angle brackets around the stereotype name).

Besides the element and compartment types, every graph diagram type can have an associated toolbar consisting of toolbar elements. We consider only pre-defined (core) toolbar elements whose implementation is provided by GDE in Core TDMM.

The graph diagram type has an associated palette to be shown with particular diagrams. Each of the palette elements are connected to one or more (in case of ports or lines) element types. This connection determines the type of element being created when a palette element is activated. If several line or port types are connected to one palette element (for instance, in class diagrams it may be convenient to use the same palette element for creating associations and links), the type of element is determined by the context of the corresponding *NewLineEvent* or *NewPortEvent*. If there is more than one possible alternative, the list of options is presented to the user.

The context menus (*ContextMenu* instances) can be ascribed to element types as well as to graph diagram types. There may be different context menus for the same diagram depending on the existence of selected elements in the diagram; therefore, there are two associations – *contextCollection* and *contextEmpty* – from *GraphDiagramType* to *ContextMenu*. In Core TDMM we consider only items implemented by GDE (symbol style), or that are provided a universal implementation on the level of tool definition platform ("properties", "copy", "cut", "paste", "delete", "refine").

Similarly, we include a keyboard with universal keys in Core TDMM, allowing for standard editor functionality (e.g., Ctrl+C for "copy", Ctrl+V for "paste", etc), or serving as shortcuts for GDE services (e.g., Ctrl+> for "zoom in" etc).

Implementation of the tool definition framework is achieved by developing an interpreter that, relying on the existing implementation of GDE (Section 2), interprets a particular instance of TDMM in the way the corresponding tool reacts from the end user's point of view.

Regarding semantics of Core TDMM and its interpreter, we note that *LClickEvent* does not invoke a transformation, *RClickEvent* prepares and opens context menu (via *ActivateContextMenuCmd*), and *L2ClickEvent* opens a property dialog.

The interpreter also uses a property dialog engine (PDE) with a metamodel-based interface (the property dialog metamodel, PDMM). This architecture allows the interpreter to be written as a collection of model transformations. The transformations have been created for all events of GDE, and they are responsible for the "business logic" of the tool that corresponds to the semantics of Core TDMM, outlined here. We have used the model transformation language L0+ [17] for our implementation; however, other "higher-level" transformation languages could have been used as well (e.g., the graphical model transformation language MOLA [18]).

An alternative approach to particular tool definition could be to write the transformations implementing the behavior of the tool directly against the events of GDMM. The possibility remains to replace some of the platform-defined transformations by tool-specific transformations (for instance, one may replace the "properties" transformation by "refine" transformation (navigate from the seed to the child) as a response to *L2ClickEvent* for some specific element types). Our approach to introducing tool-specific behavior (explained in Section 4), however, is via a mechanism of extending universal platform-level transformations instead of replacing them, so that the functionality present in the platform-level interpreter is efficiently retained.

As to the expressiveness of the proposed metamodel, a very wide range of graphical tools (inter alia an editor for EMOF [13] class diagrams and UML activity diagrams) can be defined as its instances.

We note that many popular and powerful meta-tools (see, for instance, MetaEdit [2, 3]) do not present an explicit tool definition metamodel, but explain the tool behavior by means of some configuration facilities for the end user instead. Some meta-tools provide the possibility to use more powerful constraints in some constraint definition language. However, if we want to offer a really dynamic behavior, we have to do serious programming and to understand the implementation of the particular meta-tool thoroughly. In our approach, all information relevant to DST building and running is captured as an instance of an expressive yet sufficiently simple metamodel (Fig. 2), thus providing sufficiently easy means for tool functionality extensions. These extension opportunities are described in the next section.

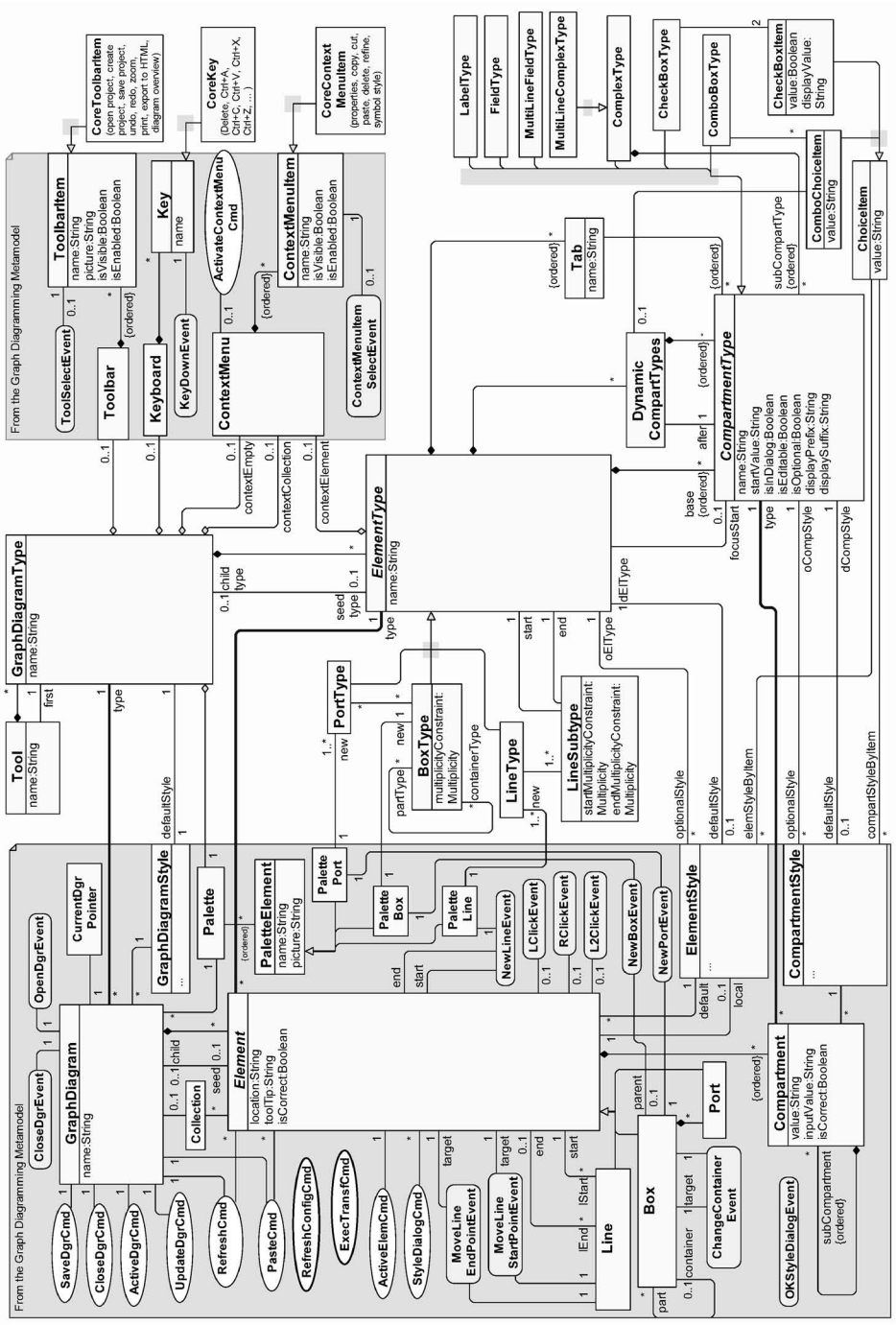

Fig. 2. The tool definition metamodel

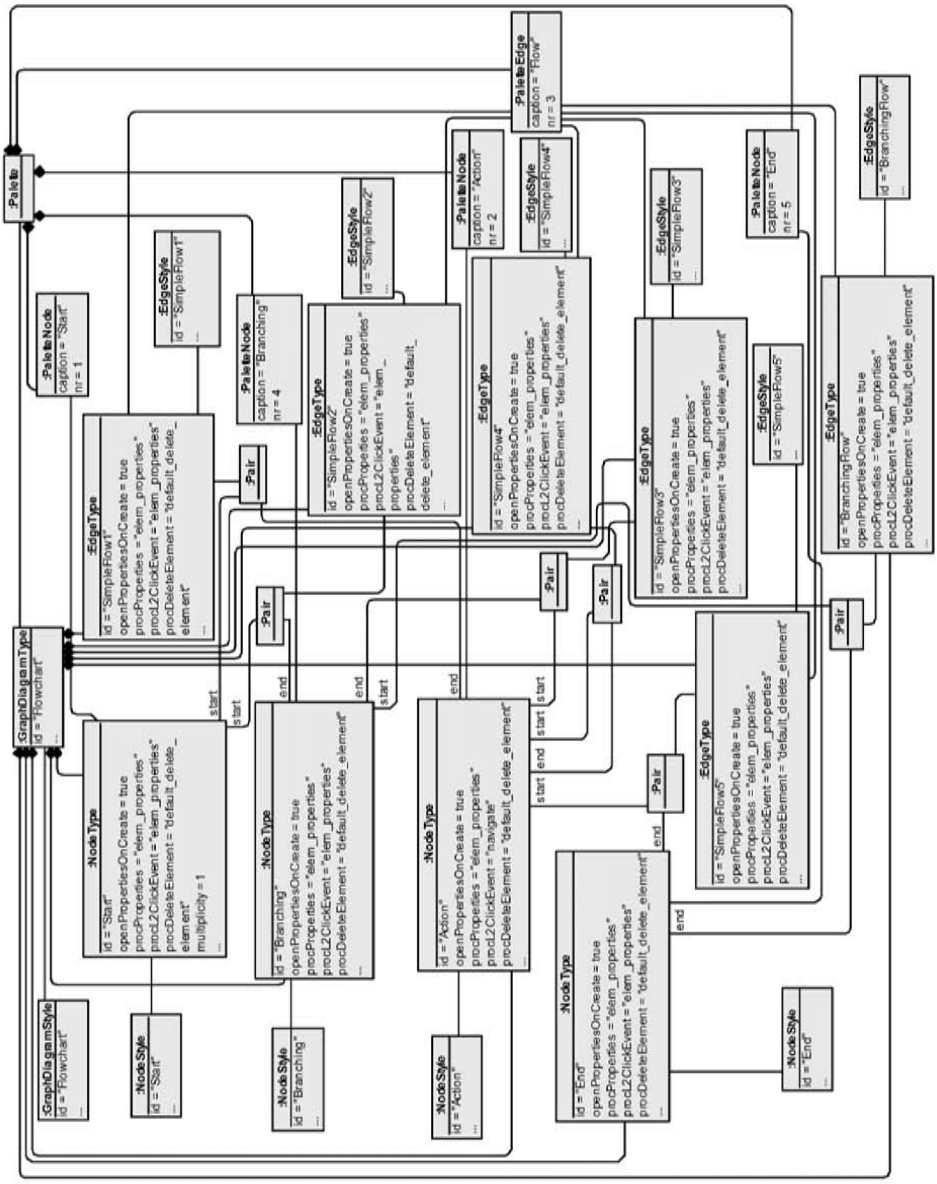

Fig. 3. Graph diagram type "Flowchart" and its context

3.1 Instantiation of the Tool Definition Metamodel

In this sub-section, an example is given in a form of a simple flowchart editor, which is an instance of the tool definition metamodel. Since the instance graph turned out to be quite large and thus unreadable for humans, it has been divided into three parts here. The first part (Fig. 3) contains the top level type information – instance of *GraphDiagramType* representing the flowchart diagram type – and its context. Here, any Flowchart diagram

is to consist of four *NodeTypes* – Start, End, Action, and Branching. For each of them, the most important attributes are given. For example, for the Start element, it is said that only one element of this type is allowed in a flowchart (see attribute "multiplicity"). Also, some transformation names can be seen, e.g., a transformation "navigate" is to be called when user double-clicks an Action, while a transformation "elem_properties" is to be called when user double-clicks a node of some other type. Next, several *EdgeTypes* exist in order to offer an opportunity to draw a line between nodes of different types. However, all these edge types are connected to one *PaletteEdge* called "Flow"; thus, the user is not responsible for picking the right palette element for different flow types – they all look alike from the user's point of view. Finally, *NodeStyle* and *EdgeStyle* instances are present as well. Due to the large number, all style attributes are not listed here.

The second part of the instance graph contains detailed information about the four node types sketched in the first part (Fig. 4 and 5). For every node type, a *PropertyDiagram* and a *PopUpDiagram* is depicted. The property diagram is a way to specify the property dialog window to be opened when the user, for example, double-clicks some element. Here, property diagrams of Action and Branching node types each consist of one *PropertyRow* being a simple text field (see attribute "rowType") for entering and altering the values of the respective compartments (of *CompartmentTypes* "Expression" and "Condition", respectively). The pop-up diagram contains *PopUpElements* to be shown

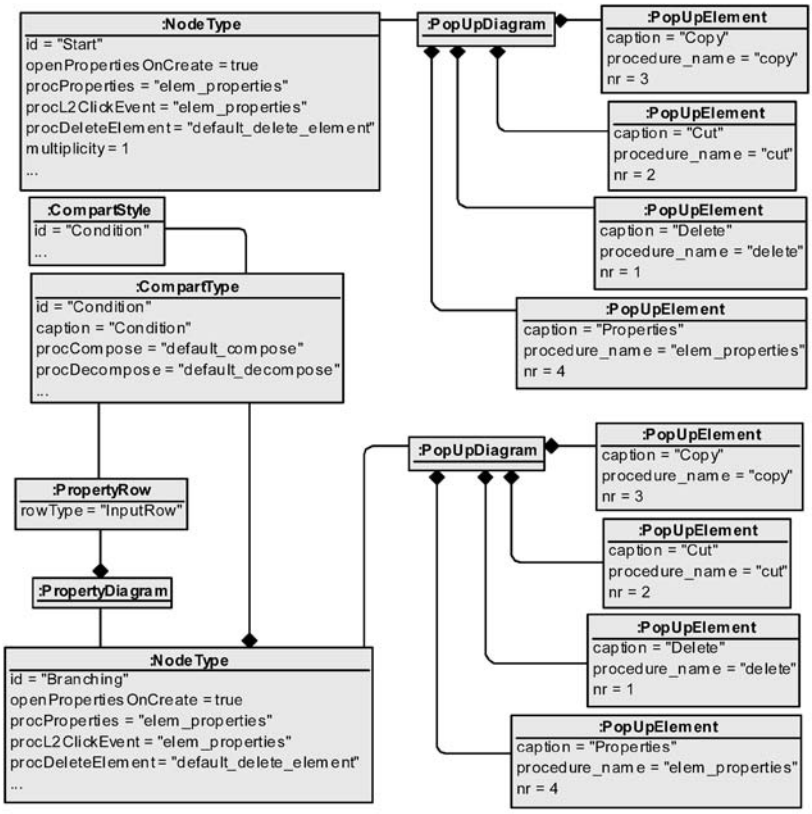

Fig. 4. Flowchart node types "Start" and "Branching" and their context

when the user, for example, clicks with the right mouse button on some element. Here, each pop-up menu contains four elements for standard actions “copy”, “cut”, “delete”, and “properties”. For each pop-up element, a calling transformation name is specified with the attribute “procedure_name”.

The third part of the instance graph contains detailed information on the edge types sketched in the first part (Fig. 6). The information to be specified for an edge type is approx. the same that needs to be specified for a node type. Thus, instances shown here are quite alike to those shown in Fig. 4 and 5.

4 The Tool Definition Metamodel: Extensions

The implementation of Core TDMM, as described in Section 3, attached a fixed universal model transformation to every event of the presentation engine (GDE). However, there may be situations in advanced tool building when such standard universal functionality is not sufficient and a tool-specific behavior is required. For instance, there may be a need to synchronize the contents of the graphical editor with data in some other source (e.g., a domain model), or there may be some further restrictions or constraints to be observed regarding elements and values that can be introduced during the diagram editing process.

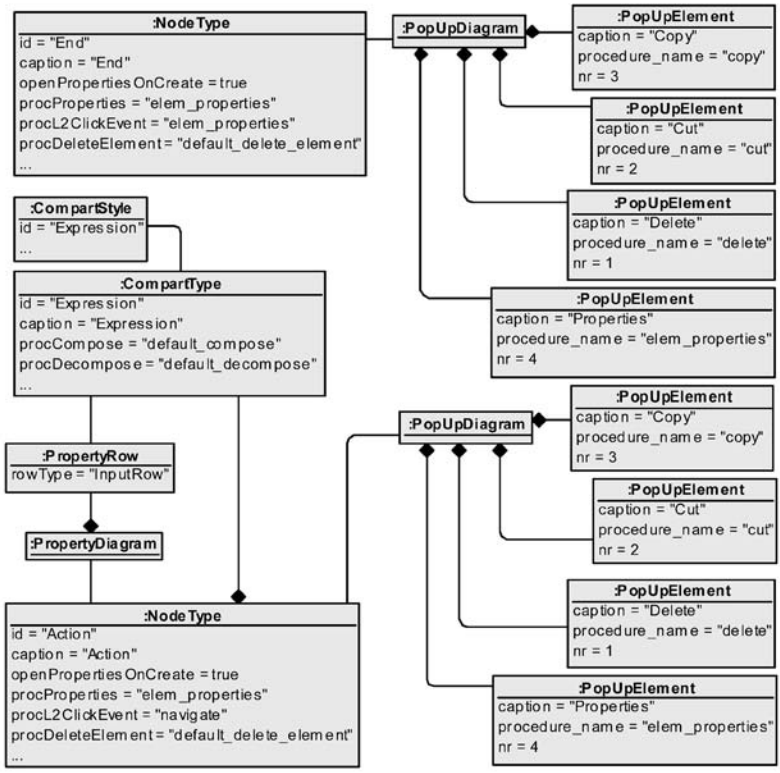

Fig. 5. Flowchart node types “End” and “Action” and their context

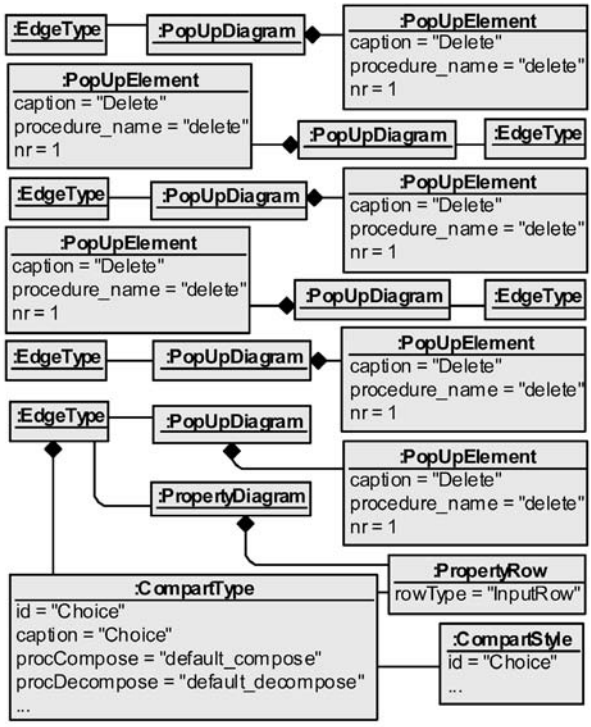

Fig. 6. Flowchart edge types and their context

Since the tool to be defined by the tool definition platform conforms to the given tool definition metamodel, in principle, it is possible to allow the tool builder to write his/her own model transformations for handling certain events instead of the transformations in-built in the platform (these transformations can work in terms of the tool definition metamodel). However, our approach to the tool functionality extension is more refined in that we allow the tool builder willing to introduce the extended functionality to rely on the basic work done by the transformations implementing the platform nevertheless. This is achieved by extending Core TDMM with classes *XElemType* and *XCompartType* that are subclasses of *ElemType* and *CompartType,* respectively (Fig. 7). These classes contain attributes that correspond to certain call points at which the platform-level event processing transformation (which is to be adopted to respect these call points) may give control over to an external tool-specific transformation.

The extended tool definition metamodel also contains classes *AdvancedKey*, *AdvancedContextMenuItem* and *AdvancedToolBarItem* that provide the tool constructor with more points where the tool-specific transformations can be attached.

In the remaining paper, we explain the semantics of particular call points – their placement in the tool interpretation process. We claim that this explanation, together with understanding of the tool definition metamodel, is sufficient to efficiently use the call point mechanism in advanced DST building. This is in sharp contrast with the amount of platform-specific implementation details required for developing advanced tools, for instance, in the Eclipse GMF platform [8].

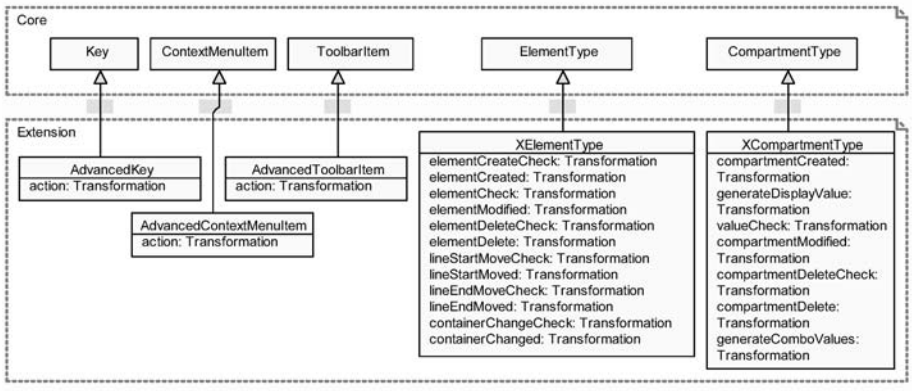

Fig. 7. The tool definition metamodel: extensions

Table 2 summarizes the call points in the *XElementType* class that arise in connection with element creation, content modification and deletion (if not specified otherwise, each transformation accepts a corresponding instance E:*Element* as its only argument; the call points are designed to have transformations that either do or do not have a (Boolean) return value).

Table 2

Call points in *XElementType*

elementCreateCheck : Boolean	Called before creating an element (an instance of the *Element* class). If the function returns **false**, the element creation process is canceled. Recommended for initial correctness constraints (e.g., whether a new element of the given type is possible in the diagram).
elementCreated	Called after creating the element, after *elementCreateCheck*, before adding compartments.
elementCheck : Boolean	Called upon completing value change of the element's compartments. The result of the function is recorded in the element's *isCorrect* attribute. The user is notified if the transformation returns **false**.
elementModified	Called upon completing value change of the element's compartments, after *elementCheck*.
elementDeleteCheck : Boolean	Called upon the user's request to delete an element, after the system's own checks for the possibility to delete are completed. If the return value is false, the "delete" action is canceled.
elementDelete	Called upon the user's request to delete an element, after *elementDeleteCheck*, before (unconditional) deleting of the element.
lineStartMoveCheck (E, OLDSTART, NEWSTART: *Element*): Boolean	Called upon the user's request to move the line's start point, after the system's own checks for the possibility of action are completed. If the procedure returns **false**, the action is canceled.
lineStartMoved (E, OLDSTART, NEWSTART: *Element*)	Called after the line's start point has been moved.
lineEndMoveCheck (E, OLDEND, NEWEND: *Element*): Boolean	Called upon the user's request to move the line's end point, after the system's own checks for the possibility of action are completed. If the procedure returns **false**, the action is canceled.

lineEndMoved (E, OldEnd, NewEnd: *Element*)	Called after the line's end point has been moved.
containerChangeCheck (E:*Element*,OC:[*Element*], NC:[*Element*]) : Boolean	Called upon the user's request to change the element's container (e.g., to move a box in or out another box, or from one containing box to another), after the system's own checks for the possibility of action are completed. [*Element*] denotes optional argument of type *Element*. If the procedure returns **false**, the action is canceled.
containerChanged(E:*Element*,OC:[*Element*], NC:[*Element*])	Called after the element's container has been changed.

Note. Moving a line's start or end point or changing a container do not invoke initial deletion and further creation of elements; therefore, the corresponding call points for element deletion and element and compartment creation are not activated.

Table 3 summarizes the call points in the *XCompartmentType* class (each transformation accepts a corresponding instance C:*Compartment* as its argument).

Table 3

Call points in *XCompartmentType*

compartmentCreated	Called after creating compartment and setting its context (link to the element or containing compartment), before setting up the compartment's value and processing sub-compartments.
generateDisplayValue	If specified, is used instead of the Core mechanisms for generating the compartment's value (as seen in the diagram) from an input value (as entered in the property editor). Called after the input value of the compartment is prepared (e.g., in the property editor).
valueCheck: Boolean	Called upon completing a value change of the compartment, after *generateDisplayValue*. The result of the procedure is recorded in the compartment's *isCorrect* attribute. The user is notified if the transformation returns **false**.
compartmentModified	Called upon completing a value change of the compartment, after *valueCheck*.
compartmentDeleteCheck : Boolean	Called upon the user's request to delete the compartment, after the system's own checks for possibility to delete are completed. If the procedure returns **false**, the action is canceled.
compartmentDelete	Called upon the user's request to delete the compartment, after *compartmentDeleteCheck*, before (unconditional) deleting of the compartment.
generateComboValues	Procedure for dynamic generation of values in the compartment's combo box in the property editor. If unspecified, the combo box is filled up by means specified in the Core.

Note. The *compartmentDeleteCheck* and *compartmentDelete* transformations are not called when deleting a whole element.

Note 2. The tool-specific transformations inserted at the call points are not automatically invoked in case of the user's own manipulation of the model contents behind the platform's event-processing transformations.

The introduced tool extension mechanism, albeit simple, is sufficient for a large range of tasks arising in DST building. We mention some of them here:

- synchronization with an abstract user-defined domain model,
- constraints of potentially arbitrary logical complexity,
- dynamic contents in the tool (e.g., drop-down values in a combo-box),
- advanced dependencies in the tool's presentation behavior,
- integration with other data engines (e.g., data from relational databases, provided the data access interface is created).

Synchronization of the contents of the model with a user-defined domain model can be performed by transformations *elementCreated*, *elementModified* and *elementDelete*, as well as *compartmentCreated*, *compartmentModified* and *compartmentDelete* that provide the tool builder the points at which a corresponding action can be defined in the domain model (e.g., creating, modifying or deleting a structure corresponding to an element or compartment on the presentation level). If necessary, the *lineStartMoved*, *lineEndMoved* and *containerChanged* transformations can also be used for this purpose.

The constraints can be implemented in the tool by the transformations *elementCreateCheck*, *elementCheck*, *elementDeleteCheck*, *lineStartMoveCheck*, *lineEndMoveCheck*, *containerChangeCheck*, *compartmentDeleteCheck* and *valueCheck*. All these transformations, except *elementCheck* and *valueCheck*, cancel the action initiated by the user in case of returning **false**. The result of *elementCheck* and *valueCheck* transformations is placed in the element's or compartment's attribute *isCorrect*, and the user is notified to take a correcting action in the case if the result had been false. Note that both the structure of the model created in the editor (the presentation) and the tool-specific domain model information can be accessed by the procedures implementing the constraints.

Since the DST conforms to the (extended) tool definition metamodel (is an instance of this metamodel), the transformations attached to the call points as well as the event-processing transformations defined by the user (in case of *AdvancedKey*, *AdvancedContextMenuItem* and *AdvancedToolBarItem*) can be defined, in principle, in any high-level model transformation language. This means that we have reached a point when an advanced DST including user-defined extensions can be fully implemented within an MDE framework without the need to resort to structures and constructs typical of programming languages. With the extension mechanism, programmers are free to add a dynamic behavior to the tool being created without putting in too much effort. The simplest example is perhaps generation of combo box items dynamically – if needed, the transformation *generateDisplayValue* can do the job.

Furthermore, the definition of the call points in the tool interpretation process hides the details of the tool interpretation process from the user (it allows the user to seamlessly re-use the implemented process). It allows the user to focus just on adding the tool-specific advanced functionality and rely on the fact that transformations will be called at the right time and place. The only requirement for the tool builder (the writer of extension transformations) is not to introduce inconsistencies in the metamodel depicted in Fig. 2.

5 Conclusions

In this paper, we have presented a universal tool definition metamodel with an extension mechanism that allows us to construct advanced DSTs while staying within the MDE framework. The static part of the tool has to be first defined as an instance of the (extended) tool definition metamodel, and then the model transformations for tool-specific operations as well as for defined call points can be provided.

The definition of the static part of the tool can be performed by a model transformation, setting up the appropriate instances necessary for the work of the tool (these include instances of *ElementType*, *CompartmentType*, as well as *ElementStyle* and *CompartmentStyle* and their related classes). Nevertheless, our implementation of the platform also provides a configurator (as most of DST building platforms do) that can be used to set up the tool's instances in a user-friendly way. All our test cases (including the UML 2.0 class diagram editor with a full support of attributes, stereotypes and tagged values) and practical applications of the platform (including several document flow and workflow modeling systems, e.g. [19]), have been successfully created using the configurator and providing the specific transformations at suitable extension points, where necessary.

We note that for a large range of tools, most of the tool functionality fits into Core TDMM and that the transformations at the call points tend to be rather small in size. We usually call them "mini-transformations"; however, we also recognize the potential of using more powerful transformations.

The MDE-based platform has allowed for building of modeling tools that are integrally incorporated into larger business information infrastructure where the graphical modeling of processes within a DST is coupled with the organization's actual data residing, for instance, in a relational database (e.g., a transformation looking up values for a combo box drop-down list can be easily redirected to an external data source, or a copy of the system model can be easily transferred to a database where further analysis of it can be enabled, etc).

The tool architecture allows for both accessing the external data from the tool's environment (provided suitable adapters for external data are created; we have elaborated on such architecture in [16]) and accessing the tool's repository from an external application. The easy external access to the graphical contents of the tool's model has proved useful, for instance, for visualizing feedback in the model from actually implemented systems.

In practice we also noticed that model migration between the tool and platform versions by model transformations works seamlessly from the end user's point of view.

We are looking forward to new applications of our platform within the area of integrating modeling tools within larger information infrastructures as we believe in the MDE-based approach we have chosen as the basis for DST building.

6 References

1. MetaEdit+ Workbench User's Guide, Version 4.5. Available: http://www.metacase.com/support/45/manuals/mwb/Mw.html, 2009.
2. S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008, 448 p.

3. Z. Nianping, J. Grundy, J. Hosking. *Pounamu: a meta-tool for multi-view visual language environment construction.* 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC '04), 30 September 2004, pp. 254–256.
4. J. Grundy, J. Hosking, J. Huh, K. Na-Liu Li. *Marama: an Eclipse Meta-Toolset for Generating Multi-View Environments.* ICSE '08, May 10–18, Leipzig, Germany. 2008.
5. I. Rath, D. Varro. Challenges for advanced domain-specific modeling frameworks. *Proc. of Workshop on Domain-Specific Program Development (DSPD).* ECOOP 2006, France.
6. C. Ermel, K. Ehrig, G. Taentzer, E. Weiss. Object-Oriented and Rule-Based Design of Visual Languages Using Tiger. *Proceedings of GraBaTs '06,* 2006, p. 12.
7. 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.
8. Graphical Modeling Framework (GMF, Eclipse Modeling Subproject). Available: http://www.eclipse.org/gmf/.
9. S. Cook, G. Jones, S. Kent, A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007, 524 p.
10. J. W. Yoder, F. Balaguer, R. Johnson. Architecture and Design of Adaptive Object-Models. ACM SIGPLAN Notices, Vol. 36, 2001, pp. 50–60.
11. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proc. of Workshop on Model Driven Development of Advanced User Interfaces (IUI 2009).* Florida, USA.
12. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis. GrTP: Transformation-Based Graphical Tool Building Platform. *Proc. of MODELS 2007 Workshop on Model-Driven Development of Advanced User Interfaces (MDDAUI 2007).* Nashville, USA.
13. Meta-Object Facility (MOF). Available: http://www.omg.org/mof/.
14. K. Freivalds, P. Kikusts. Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. *Proc. of Latvian Academy of Sciences*, Section B, Vol. 55, No. 1, 2001, pp. 43–51.
15. P. Kikusts, P. Rucevskis. Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors. *Proc. of Graph Drawing '95*, Lecture Notes in Computer Science, Vol. 1027, 1996, pp. 361–364.
16. J. Barzdins, S. Kozlovics, E. Rencis. The Transformation- Driven Architecture. *Proceedings of DSM '08 Workshop of OOPSLA 2008*, Nashville, USA, pp. 60–63.
17. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs. Model Transformation Languages and Their Implementation by Bootstrapping Method. *Pillars of Computer Science*, Lecture Notes in Computer Science, Vol. 4800. Springer-Verlag, 2008, pp. 130–145.
18. A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA. *Proceedings of MDAFA 2004*, Lecture Notes in Computer Science, Vol. 3599. Springer-Verlag, 2005, pp. 62–76.
19. 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, pp. 34–40.