SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 139–149 P.

A Graph Diagram Engine for the Transformation-Driven Architecture

Janis Barzdins, Karlis Cerans, Sergejs Kozlovics, Edgars Rencis, Andris Zarins
Institute of Mathematics and Computer Science, University of Latvia
Raina bulv. 29, LV-1459, Riga, Latvia
{janis.barzdins, karlis.cerans, sergejs.kozlovics, edgars.rencis, andris.zarins}@lumii.lv

The transformation driven architecture (TDA) is a system building (in particular, tool building) approach that is based on model transformations, interface metamodels with corresponding engines, and event/command mechanism. This paper describes a metamodel and the corresponding engine for graph diagram presentations within TDA. The facilities of the metamodel and the engine include static diagram presentations, as well as graph diagram animations.

Keywords: transformation-driven architecture, model transformations, metamodels, graph diagrams, diagram animation, modeling tools.

1 Introduction

The increasing variety of metamodel-based tools such as MetaEdit [1], Eclipse GMF [2], Microsoft DSL Tools [3], DiaGen/DiaMeta [4] and METAclipse [5] has lead to study of principles behind tool architecture. Metamodel-based tools allow domain data to be represented in a graphical form according to some (perhaps, implicit) presentation metamodel. In [6] we have developed an approach called the *Transformation-Driven Architecture* (TDA), where not just one, but several presentation metamodels are allowed. The link between domain and presentation models within a modeling tool is established by means of model transformations.

Since a presentation model is not yet the end interface that can be presented to the user, some engine is needed to construct the corresponding diagram itself from the instance of the presentation metamodel. Presentation engines form an essential part of the TDA.

Developing a presentation engine and the corresponding metamodel may be a non-trivial task yet when implemented, the corresponding engine can be reused in several tools built upon the TDA.

In this paper a metamodel for graph diagram presentations within TDA and the corresponding engine for drawing/editing graph diagrams is presented. The metamodel along with the engine is a further development based on previous authors' work [7] by fully elaborating the metamodel and putting it within the context of TDA. The graph diagram animation facilities are also newly sketched here.

The paper is organized as follows. The next section lists some ideas of the TDA and explains how the proposed Graph Diagram Engine can be integrated within the TDA Framework. In Sect. 3 the Graph Diagram Metamodel and the Graph Diagram Engine are explained. Sect. 4 presents a way of implementing animation mechanism for graph diagrams. Finally, Sect. 5 concludes the paper.

The short version of the concepts presented in this paper is published in [8]. This is an extended version of [8] and can be presented as a technical report as well.

2 The Essence of the Transformation-Driven Architecture

The Transformation-Driven Architecture [6] is a metamodel-based system (in particular, tool) building approach, where the system metamodel consists of one or more presentation metamodels served by the corresponding engines and the (optional) Domain Metamodel. There is also the Core Metamodel (fixed) with the corresponding Head Engine. Model transformations are used for linking instances of the mentioned metamodels (see Fig. 1).

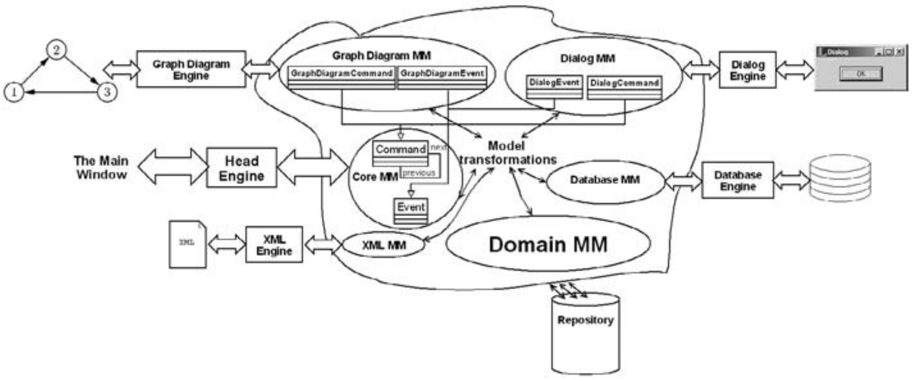


Fig. 1. Metamodels and engines in transformation-driven architecture

There is an *Event* class in the metamodel whose singleton subclasses correspond to the actions the user may perform on a particular diagram and that are understood by a number of engines. Upon observing a current event, engine invokes the event's transformation that is responsible for concrete tool's "business logic" in response to this event. The *Command* class describes the requests (commands) that the tool transformations can issue for an engine. There may be several commands issued by a single tool transformation.

The Head Engine is a special engine, whose role is to provide services for transformations as well as for presentation engines. For instance, when in a presentation engine a user event (such as a mouse click) occurs, the Head Engine may be asked to call the corresponding transformation for handling this event. A transformation may give commands to presentation engines. The Core Metamodel contains classes *Event* and *Command*, and the Head Engine is used as an event/command manager.

TDA has its own framework that comes with the built-in Head Engine (serving the Core Metamodel) and a number of predefined pluggable engines (the Graph Diagram Engine is one of them). Other presentation engines may also be written and plugged-in, as needed. The TDA framework is common to all the tools built upon the TDA. The framework is brought to life by means of transformations. One can choose between writing different transformations for different tools and writing one configurable transformation covering several tools.

3 Graph Diagram Metamodel and Graph Diagram Engine

In the course of time, the graph diagram metamodel has been evolving and providing more and more new facilities. As a result, the physical amount of metamodeling elements (classes, attributes, associations) has significantly increased and representing the whole metamodel visually is a tricky thing to do now. Therefore, in this section, the whole graph diagram metamodel is divided in several parts and each part is discussed in a separate subsection. From here on – if the role name for some association is not mentioned in a metamodel, it is assumed to be default, i.e., the same as the class name with the first letter in lower case.

The graph diagram engine is responsible for visualizing instances of the Graph Diagram Metamodel. The engine is developed on the basis of graphical engines for GRADE tools family [9]. The engine relies on advanced graph layout algorithms [10, 11] as well as effective internal diagram representation structures allowing to handle the visualization tasks efficiently even for large diagrams.

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

3.1 The Kernel of the Graph Diagram Metamodel

The visual elements of the presentation (see Fig. 2) correspond to the classes *GraphDiagram*, *Element* and *Compartment*. Every graph diagram can consist of elements of several distinct types – *Node*, *Edge*, *Port*, *FreeBox* or *FreeLine*. A port is a

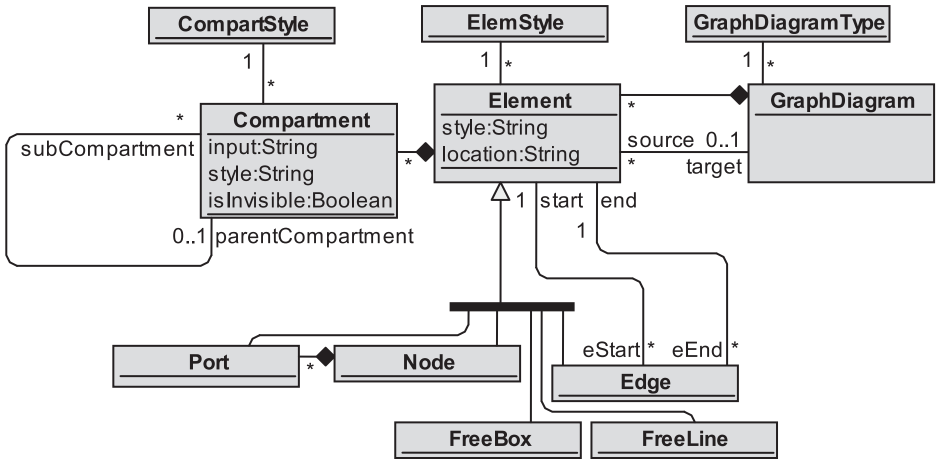


Fig. 2. The kernel of the graph diagram metamodel

small box that can not exist on its own but is instead attached to a *Node*. An edge always contains exactly one start element and one end element as noted by associations *start* and *end*. Free boxes and lines denote visual objects having no layout constraints to be satisfied by the graph diagram engine. Compartments correspond to text fields that may be placed inside nodes or attached to edges and ports. The value of the field is stored in the *input* attribute, and the compartment itself can be made invisible by changing the value of its attribute *isInvisible*.

Instances of the classes mentioned above are diagrams and graphical elements created by the user. Every element and compartment has exactly one style (see classes *ElemStyle* and *CompartStyle*) denoting the visual appearance of the element (or compartment). Instances of classes *ElemStyle* and *CompartStyle* store the default styles of elements and compartments, while the actual style is coded as a string and stored in the *style* attribute of classes *Element* and *Compartment*. Graph diagram engine generates the style string at element or compartment creation time accordingly to the style instance attached to it. It is allowed to change the actual style at runtime (by changing the *style* attribute) while the default style remains unmodified. Likewise, the *location* attribute of *Element* is generated by the graph diagram engine.

In the case of *GraphDiagram*, the class *GraphDiagramType* is attached to it containing both type and style information for the graph diagram. For classes *Element* and *Compartment*, the type information is separated from the style information by making classes *ElemType* and *CompartType* separately from classes *ElemStyle* and *CompartStyle*. The type information goes beyond the scope of this paper and thus will not be discussed in more detail here (see [12] for more details).

Navigation among diagrams can be made according to the metamodel by using the "source – target" association between *Element* and *GraphDiagram*. The other type of hierarchy is the compartment containing hierarchy implemented by the "parentCompartment – subCompartment" association.

3.2 *GraphDiagram* and Its Context

As was stated before, *GraphDiagramType* contains style information for the diagram. This information is put in attributes of the class *GraphDiagramType* (see Fig. 3). When a diagram is being made, one can copy the values of attributes to the attributes of the particular *GraphDiagram*, thus giving it the default style. These values can, however, be changed to assign an individual style to a diagram. The meaning of the style attributes is explained in the next paragraph.

First, a diagram can have a *caption* that will be seen at the title of the diagram window. Next, diagrams background color is coded in *bkgColor* and *layoutMode* and *layoutAlgorithm* imply layout information, for example whether the layout mode is automatic, semi-automatic or completely manual. Value of this attribute is coded as integer 0, 1 or 2, respectively. Finally, *screenZoom* and *printZoom* are responsible for the scale of the diagram.

Next, a set of active elements can be found in a graph diagram. Therefore, a class *Collection* is present here. The active diagram itself can be found following the link from the only instance of the singleton class *CurrentDgrPointer*.

Every *GraphDiagram* has its context defined by classes *Palette, PopUpDiagram* and *KeyboardShortcut* and is attached to the diagram through *GraphDiagramType*.

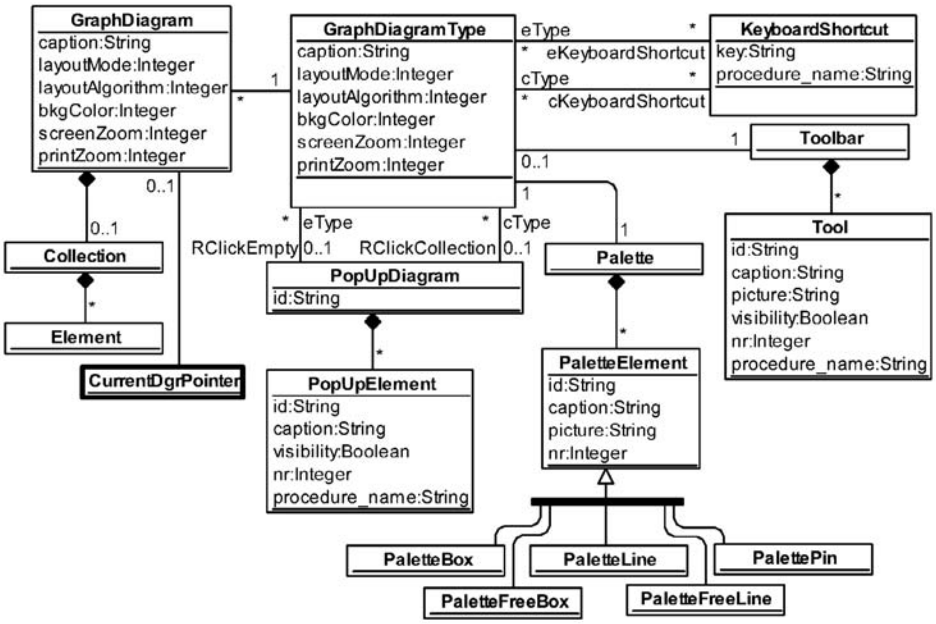


Fig. 3. GraphDiagram and its context

Palette consists of *PaletteElement*s, each of them being a line, a box, a port, a freeline or a free box. Apart from *id* and *caption*, every palette element can have a *picture* (a path to some graphical image) and an *nr* denoting the sequence in the palette.

*Toolbar*s consisting of *Tools* can also be associated with the *GraphDiagramType*. When the graph diagram is being activated, the corresponding toolbars are made visible. Like palette elements, tools can also have an *id*, a *caption*, a *picture* and an *nr*. Moreover, tools can be made invisible by setting the value of the attribute *visibility* to false. The attribute *procedure_name* must contain the name of an existing procedure to be called whenever the user presses the tool in the toolbar. It is assumed that a procedure with such a name can be found in the default dynamic link library provided in the tool (*main.dll*). If the procedure is contained in other dynamic link library than *main.dll*, the library name must be specified as well (following the syntax "<dllName>#<procedureName>").

The metamodel allows the user to specify a *PopUpDiagram* consisting of *PopUpElements*. Usually this kind of menu is activated when the user clicks the right mouse button. Depending on the context, two types of *PopUpDiagrams* can exist – one for the right click in an empty spot of the diagram, and another for the right click on a set of selected elements. Therefore, two associations between classes *GraphDiagramType* and *PopUpDiagram* exist. As it was done before for tools, a calling *procedure_name* must be specified here as well.

Finally, *KeyboardShortcuts* can be added to *GraphDiagramType* providing a possibility to perform some actions using a keyboard. Shortcuts can be specified for both cases – when a set of elements is or is not selected there. For every shortcut, a *key* and a calling *procedure_name* must be specified.

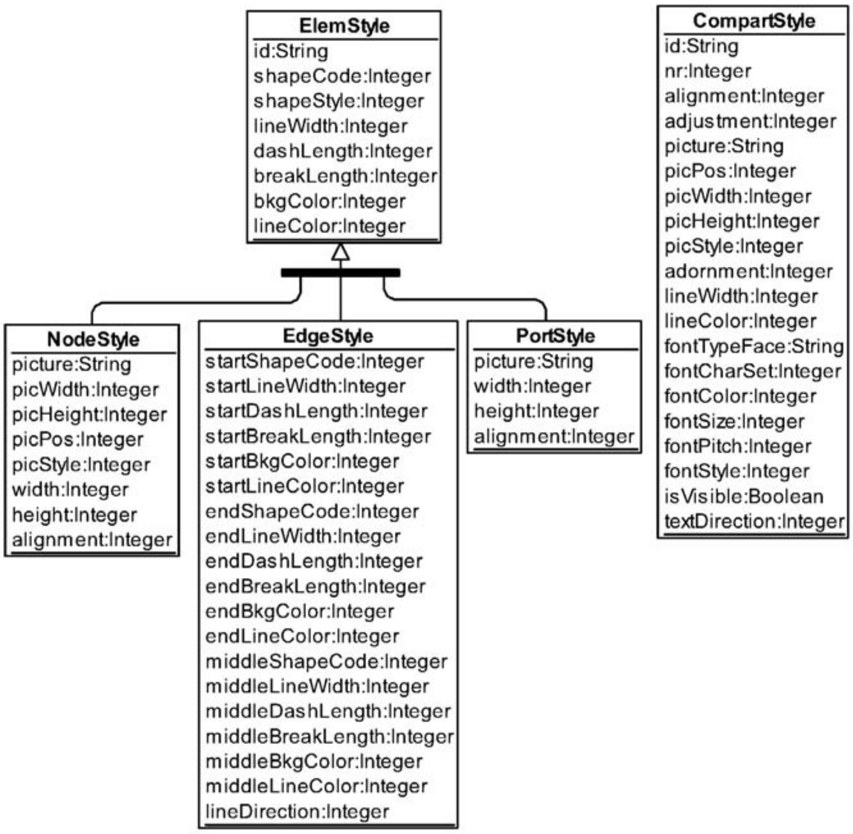


Fig. 4. Element and compartment styles

3.3 Element and Compartment Styles

As mentioned above, instances of classes ElemStyle and CompartStyle contain the default style information for elements and compartments, respectively. The style is a set of several style attributes that can be seen in Fig. 4.

Most of the Element style attribute depend on the particular Element subclass, and thus ElemStyle is divided in three subclasses as well. However, some attributes are generic enough to be attached directly to the superclass. These are *id*, *shapeCode*, *shapeStyle*, *lineWidth*, *dashLength*, *breakLength*, *bkgColor* and *lineColor*.

3.4 Events and Commands

The Graph Diagram Metamodel defines engine-specific events and commands that are subclasses of *Event* and *Command* (see Fig. 5 and 6, events and commands are white classes). Every event and every command during tool runtime is placed within the context defined by the metamodel. For example, the *NewBoxEvent* is attached to the *PaletteBox* with which it is being created, and the *Box* in which it is being put (see associations from class *NewBoxEvent*). All the events together with their context can be seen in Fig. 5, while Fig. 6. represents the commands.

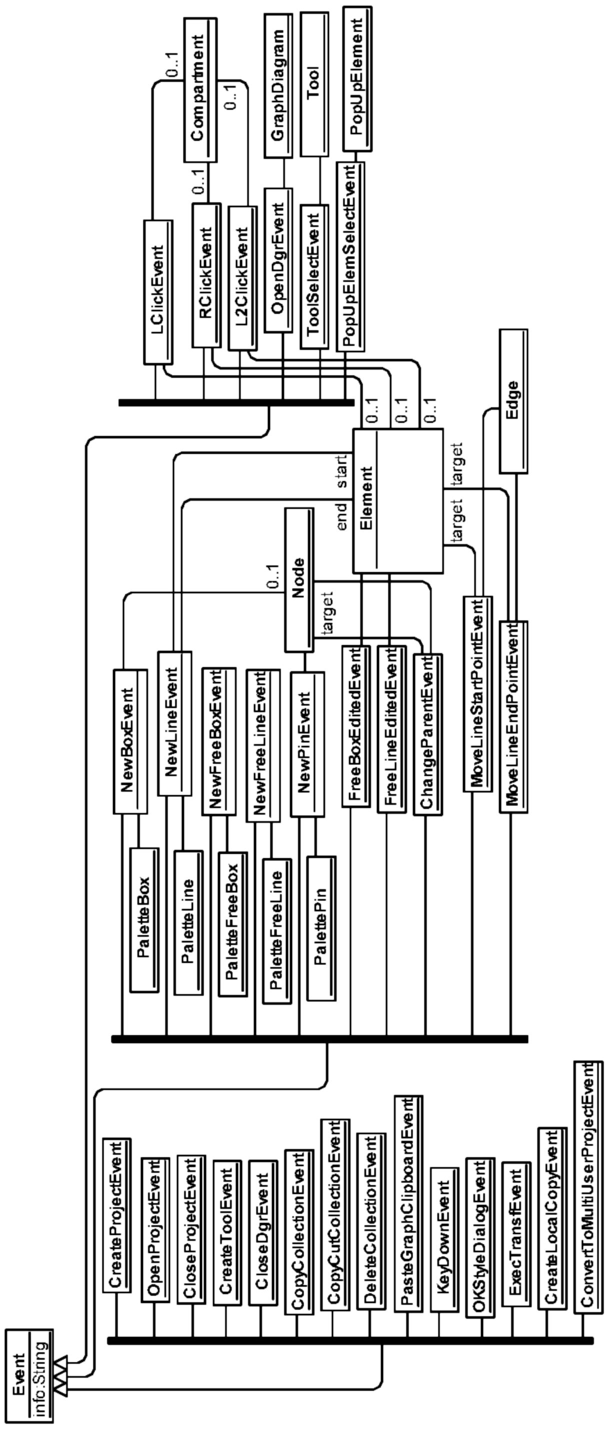


Fig. 5. Events and their context

The meaning of events and commands is mostly inferable from their names. For some events and commands, an additional attribute *info* is needed, i.e., the code of the pressed key is stored in that attribute in the case of *KeyDownEvent*. The multiplicities of

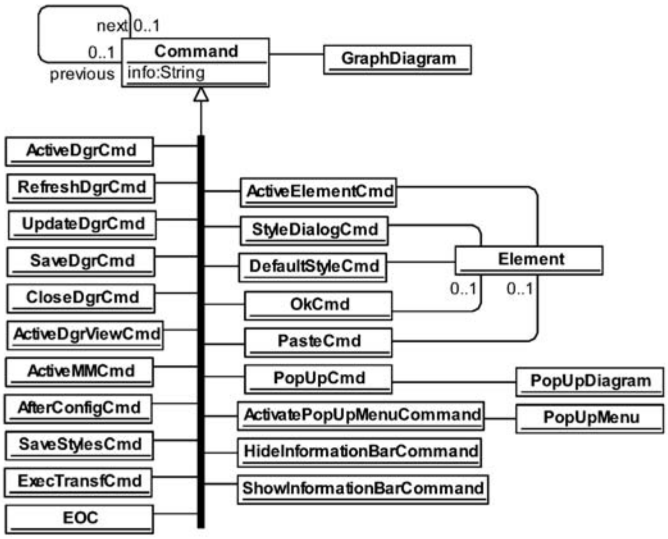
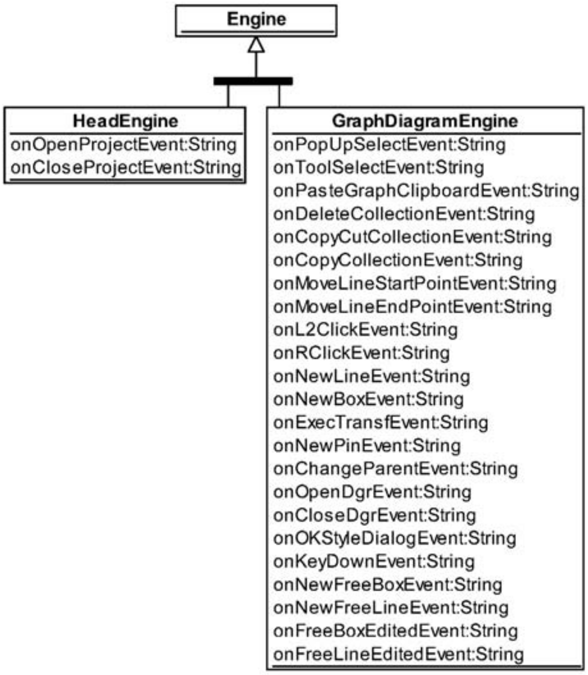


Fig. 6. Commands and their context



Engine
HeadEngine
onOpenProjectEvent:String
onCloseProjectEvent:String
GraphDiagramEngine
onPopUpSelectEvent:String
onToolSelectEvent:String
onPasteGraphClipboardEvent:String
onDeleteCollectionEvent:String
onCopyCutCollectionEvent:String
onCopyCollectionEvent:String
onMoveLineStartPointEvent:String
onMoveLineEndPointEvent:String
onL2ClickEvent:String
onRClickEvent:String
onNewLineEvent:String
onNewBoxEvent:String
onExecTransfEvent:String
onNewPinEvent:String
onChangeParentEvent:String
onOpenDgrEvent:String
onCloseDgrEvent:String
onOKStyleDialogEvent:String
onKeyDownEvent:String
onNewFreeBoxEvent:String
onNewFreeLineEvent:String
onFreeBoxEditedEvent:String
onFreeLineEditedEvent:String

Fig. 7. Engine-specific classes

roles is omitted in figures due to the similarities – the multiplicity is always "0..1" at the event side of an association, and "1" at the other side (if some role does not match the criteria, its multiplicity is noted separately).

The singleton class *GraphDiagramEngine* contains attributes that correspond to engine's events (see Fig. 7). In the beginning a transformation can assign values for these attributes, each value representing the name of the transformation that has to be called when the particular event occurs. In TDA, other singleton subclasses for other engines exist there as well. In Fig. 7, class *HeadEngine* is represented together with its attributes for its two events – *OpenProjectEvent* and *CloseProjectEvent*.

4 Graph Diagram Animation

Although there are several approaches for metamodel-based handling of dynamic multimedia objects that include animations (see, for instance, [13]), our goal here is more specific — to provide simple animation facilities for graph diagrams explained in Section 3. Complex interactive animations (such as animations that can be created in *Microsoft Silverlight* [14] or *Adobe Flash* [15]) are beyond the scope of this approach.

In Fig. 8 we extend the metamodel of graph diagrams by classes for describing graph diagram animations. The animation of graph diagrams is based on the concept of token that is associated with some element (box or line) in a graph diagram. Tokens do

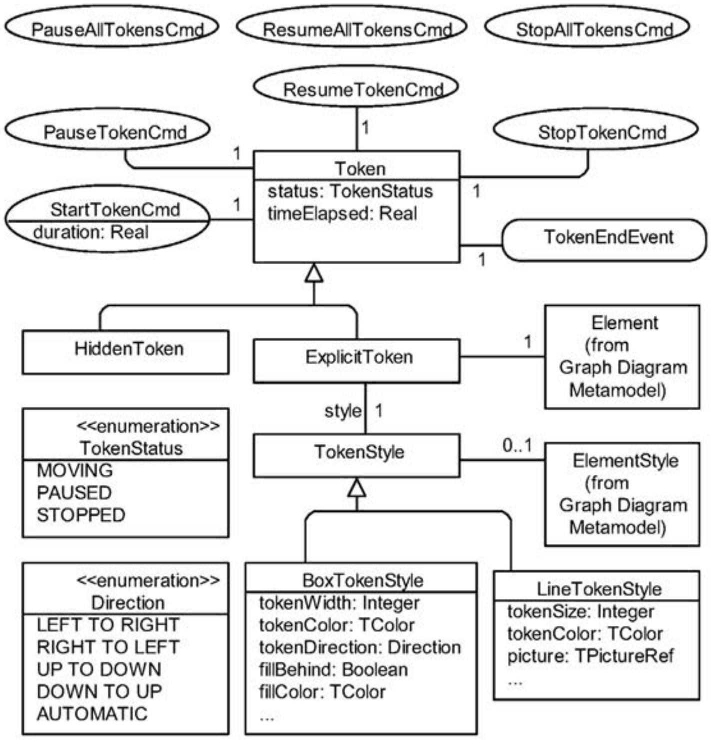


Fig. 8. Adding animation capabilities to the Graph Diagram Metamodel

not imply any semantics, they are used only for managing the animation process. The semantics is up to transformations.

A token is started by *StartTokenCmd* that also specifies its *duration* (how long the token “lives”). There are also commands for starting, stopping, pausing and resuming a token in the diagram, as well as pausing, resuming and stopping all tokens in the diagram. The “end of life” of a token is determined by the presentation engine – at that time it creates a corresponding *EndTokenEvent*. There can be several tokens living concurrently in the diagram.

An *explicit token* is able to simulate the activity of the associated element for the given duration. The visual effect of the simulation is determined by *TokenStyle* instance associated with the token. If an *ElementStyle* instance is associated with the token style, then the animation consists of changing the element style for the token’s lifetime. Other options of animation consist of moving a bullet of certain size or some image along the line in the diagram, or animating a box by a line moving across it in certain direction, with or without leaving the trailing part in the specified color. In the case of *AUTOMATIC* direction, the actual line flowing direction is determined by the presentation engine on the basis of the placement of the actual outgoing line from the box. A *hidden token* does not animate any element, it just “lives” for the specified amount of time. Hidden tokens can be useful, e.g., for accounting the global animation time, or for creating certain breakpoints during the animation when the control is transferred to transformations for some semantic actions.

The implementation of animation facilities in our graph diagram engine is currently under development.

5 Conclusions

The Graph Diagram Engine has been successfully implemented in a recent version of transformation-based tool building platform GrTP [7]. The GrTP tool is now being transformed to the TDA framework, which should become publicly available soon. At the moment, the TDA framework consists of two predefined engines (one of them is the Graph Diagram Engine and the other is the Dialog Engine), and the interaction between these engines and model transformations performed by means of commands and events is working quite well. We are working on ameliorating the TDA framework and its engines. One of the research topics here is adding advanced graph diagram layout capabilities to the Graph Diagram Engine. We are also working on implementing diagram animations within the Graph Diagram Engine for TDA.

Several diagram editors (such as class diagram editor and activity diagram editor) have been successfully built using the Graph Diagram Engine. This engine has also been used in [16] and [17]. We are looking forward for applying the TDA and its engines in the Semantic Web domain.

Acknowledgments.

The authors would like to thank (alphabetically) A. Kalnins, L. Lace, R. Liepins, A. Sprogis, R. Zarits and M. Zviedris for their efforts in implementing the concepts presented in this paper.

References

1. MetaEdit+. Available: http://www.metacase.com.
2. A. Shatalin and A. Tikhomirov. *Graphical Modeling Framework Architecture Overview.* Eclipse Modeling Symposium, 2006.
3. S. Cook, G. Jones, S. Kent, A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.
4. DiaGen/DiaMeta. Available: http://www.unibw.de/inf2/DiaGen.
5. 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.
6. J. Barzdins, S. Kozlovics, E. Rencis. The Transformation-Driven Architecture. *Proceedings of DSM'08 Workshop of OOPSLA 2008*. Nashville, USA, 2008, pp. 60–63.
7. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis. GrTP: Transformation-Based Graphical Tool Building Platform. *Proceedings of MDDAUI Workshop of MoDELS 2007*. Nashville, USA, 2007.
8. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proceedings of MDDAUI'09 Workshop of International Conference on Intelligent User Interfaces 2009*, Sanibel Island, Florida, USA, 2009, pp. 29–32.
9. GRADE tools. Available: http://www.gradetools.com.
10. P. Kikusts, P. Rucevskis. Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors. *Proceedings of Graph Drawing '95*, LNCS, vol. 1027, Springer-Verlag, 1996, pp. 361–364.
11. K. Freivalds, P. Kikusts. Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. *Proceedings of Latvian Academy of Sciences*, Section B, vol. 55, no. 1, 2001, pp. 43–51.
12. J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, Z. Zarins. An MDE-Based Graphical Tool Building Framework. This publication, pp 121–139.
13. A. Pleuss, A. Vitzthum, H. Hussmann. Integrating Heterogeneous Tools into Model-Centric Development of Interactive Application. *MoDELS 2007*, LNCS, vol. 4735, Springer-Verlag, 2007, pp. 241–355.
14. Silverlight Animation Overwiew. *MSDN*, Microsoft Corp. Available: http://msdn.microsoft.com/en-us/library/cc189019(vs.95).aspx.
15. Adobe Flash. Available: http://www.adobe.com/products/flash.
16. G. Barzdins, E. Liepins, M. Veilande, M. Zviedris. Semantic Latvia Approach in the Medical Domain. In: H-M. Haav, A. Kalja (eds.), *Proceedings of the 8th International Baltic Conference (Baltic DB & IS2008)*. June 2–5, Tallin, Estonia. Tallinn University of Technology Press, 2008, pp. 89–102.
17. 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.