GrTP: Transformation Based Graphical Tool Building Platform

Jānis Bārzdiņš, Andris Zariņš, Kārlis Čerāns, Audris Kalniņš,
Edgars Rencis, Lelde Lāce, Renārs Liepiņš and Artūrs Sproģis
Institute of Mathematics and Computer Science, University of Latvia
Raina bulvaris 29
Riga, LV-1459, LATVIA

{Janis.Barzdins, Andris.Zarins, Karlis.Cerans, Audris.Kalnins,
Edgars.Rencis, Lelde.Lace, Renars.Liepins, Arturs.Sprogis}@mii.lu.lv

ABSTRACT

In this paper we introduce a novel wide-profile graphical tool-building platform GrTP which is based on the principle of separating semantic domain model processing and user interface components. This principle is enabled in GrTP by a meta-model for visual information presentation ("interface meta-model"). A novel feature of GrTP is that it consistently bases all work of connecting the domain and interface models on model transformations. These transformations can be produced for different domain models, thus obtaining concrete tools tailored to specific domains within GrTP. The visualization component of GrTP is based on an original high-performance graphical diagram presentation engine which embodies advanced graph drawing algorithms. The paper explains the GrTP platform along with the first practical experience of its usage.

Keywords

Tool-building platforms, model transformations, interface meta-models, domain meta-models.

1.INTRODUCTION

Increasingly more information is stored in repositories as instances of meta-models (we will call them domain meta-models), e.g. semantic web ontologies and UML diagrams. This information must be viewed and edited by diverse types of users with different needs. Usually each group requires a different view of underlying meta-model that is tailored to their needs. Generally, graph diagrams provide a convenient way to view this information. We propose a new framework GrTP for building such interfaces.

The basic difference between our proposed framework and traditional tool building platforms such as Eclipse GMF [7, 17], Microsoft DSL [1], MetaEdit+ [13, 14], Pounamu/ Marama [8, 18], Diagen/Diameta [2, 3], GME [5, 11] is in the principles of the general architecture of the framework. Let us explain this in some detail, using Eclipse GMF as a typical example. According to the GMF approach [7], besides the EMOF meta-model for defining ECore models (domain meta-models), the following three meta-models are used: the Graphical definition meta-model for defining types of graphical presentation elements, the Tooling meta-model and the Mapping meta-model, by means of which mappings between the previous ones can be defined. The specification of a concrete tool is done by building instances of all the abovementioned meta-models. The next step is the generation of Java classes (C# classes in case of MS DSL); by compiling these classes an executable tool is obtained.

This architecture is well suited in cases when the domain and graphical notation meta-models have sufficiently similar structure. In these cases the above described static mapping facilities are sufficient to define the tool functionality in an adequate way. However, frequently the domain and its graphical presentation are not so similar, and the only option is a manual code extension in Java (or C#, respectively), that requires a deep understanding of the generated code.

A question arises, whether it is possible to build a very simple (both from the architecture and user point of view) platform, where the only link between the domain and interface meta-models is by means of model transformations. This would allow describing connections of any complexity between domain and interface meta-models in a uniform and structured way.

At the core of our approach is an interface meta-model of a principally new kind, which directly contains both the classes of graphical presentation elements themselves and their types, as well as classes of events and commands. In other words, our interface meta-model describes all possible graph diagram "screenshots" and all possible end user actions (events). The presentation engine, the main control unit of GrTP, is in a sense an interpreter of interface models that are instances of the abovementioned interface meta-model. It is capable to visualise elements in accordance with their types, as well as to call transformations as responses to end user actions.

2.OVERVIEW OF GrTP

Figure 1 schematically depicts the structure of GrTP. GrTP consists of control unit, called GrTP Engine (or *Presentation Engine*, or *PE*, for short), and repository. The repository contains a fixed interface meta-model (IMM) and its instances that are interface models, as well as a domain meta-model (DMM) and its instances that are domain models (e.g., concrete UML activity diagrams that correspond to the UML activity diagram meta-model). Each interface model contains a graphical view reflecting the corresponding domain model, plus it contains instances of IMM classes that correspond to symbol palette and its elements, toolbars and popup menus, as well as user actions (called *events*) on concrete interface elements.

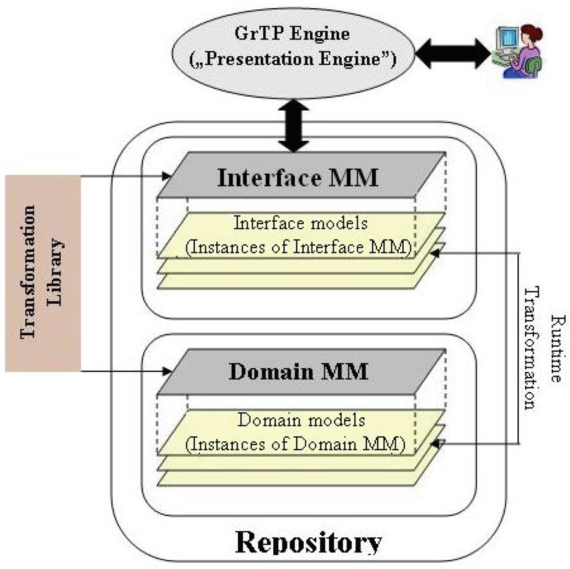

Fig. 1. The structure of GrTP

The work of GrTP Engine on a concrete DMM is powered by model transformations connecting DMM and IMM. To enable writing of these transformations, *scaffolding* between DMM and IMM that contains interconnection links of concrete DM and IM elements is used. The transformations can be written in any universal model transformation language (e.g., QVT [12], MOLA [13]) that has an efficient implementation. They are called by GrTP Engine at runtime.

The GrTP platform itself consists of a *core part* that is the presentation engine, the implementation of the repository and the IMM, and the *specialization part* that allows tailoring GrTP to the needs of serving concrete DMM. More precisely, specialization of GrTP to serve concrete DMM (that is, building of particular tool on the basis of GrTP) consists of the following:

1. providing the domain meta-model itself;
2. defining types of visual presentation elements (nodes and edges), as well as symbol palette elements (these are defined in IM as instances of respective IMM classes);
3. defining scaffolding between the elements (classes) of DMM and IMM;
4. creating the transformation library (one can think of it as of *Transformation Engine* (TE)) and connecting the transformations to the concrete events of presentation engine;

The work of the specialized tool can be perceived as cooperation between PE and TE. PE detects user actions, creates corresponding event instance and calls TE. The transformations, on the other hand, are processing the events created by PE. This usually involves making changes to instances of IMM and DMM. Transformations may also issue commands for PE that are visualization instructions to be executed by PE. We note that both Event and Command classes are included in IMM, and their instances can be linked to instances of other IMM classes.

3.INTERFACE META-MODEL

The interface meta-model is the core of the framework. It can be viewed as the interface between the presentation engine (hard-coded part) and transformations (adapted for a particular tool). Here we consider a simplified version of the interface meta-model (subset of the full interface meta-model) depicted in figure 2. We let the role names of associations, if not explicitly specified, to coincide with the corresponding class names, with first letter in lowercase.

The visual elements of the interface correspond to the classes *GraphDiagr*, *Node* and *Edge*, together with *NCompartment* and *ECompartment* that correspond to text fields placed in nodes and attached to edges, respectively. The *location* attributes in *Node* and *Edge* classes encode the placement of corresponding nodes and edges in the diagram, and they are to be understood only by the Presentation Engine; they are not meant for use by model transformations.

The IMM includes *GraphDiagrType* class for different graph diagram types (e.g., class diagrams, or activity diagrams) together with *NodeType* and *EdgeType* classes for types of nodes and edges that are allowed in diagrams of respective type. The type information for node or edge contains style information of respective element, as well as compartment type information (*NCompartmentType* and *ECompartmentType* classes). In this simplified IMM we show only shape information for nodes and both edge ends in their types (*nodeShape*, *startShape* and *endShape* attributes for simplicity are treated here as integers). The compartment type information associated with node and edge types list names of compartments that are in elements (nodes or edges) of corresponding types. The ordering of node compartments is given by the values of *nr* attribute. For edge compartments the position attribute denotes the placement of corresponding compartment (e.g. at the beginning or end of the edge). The instances of *GraphDiagrType*, *NodeType*, *EdgeType*, *NCompartmentType* and *ECompartmentType* classes, as well as instances of *Palette* and *PaletteElement* classes are created at initialization time of concrete specialized tool. These instances determine, to a large extent, the functionality of the concrete tool that is available to the end user. For instance, the collection of compartment types for a node or edge type determines the properties of compartment structure (e.g. compartment count and their placement) for nodes or edges associated with this type.

The class *Event* (▢) and its subclasses correspond to the actions that the end user may perform on a concrete diagram and that are understood by PE. The instances of these classes (called *events*) are created by PE upon detection of corresponding end user action.

The Transformation Engine, upon receiving control from PE, analyzes the type (class) of the available event, and invokes appropriate transformation, which can access the context of the event via the association links created by PE.

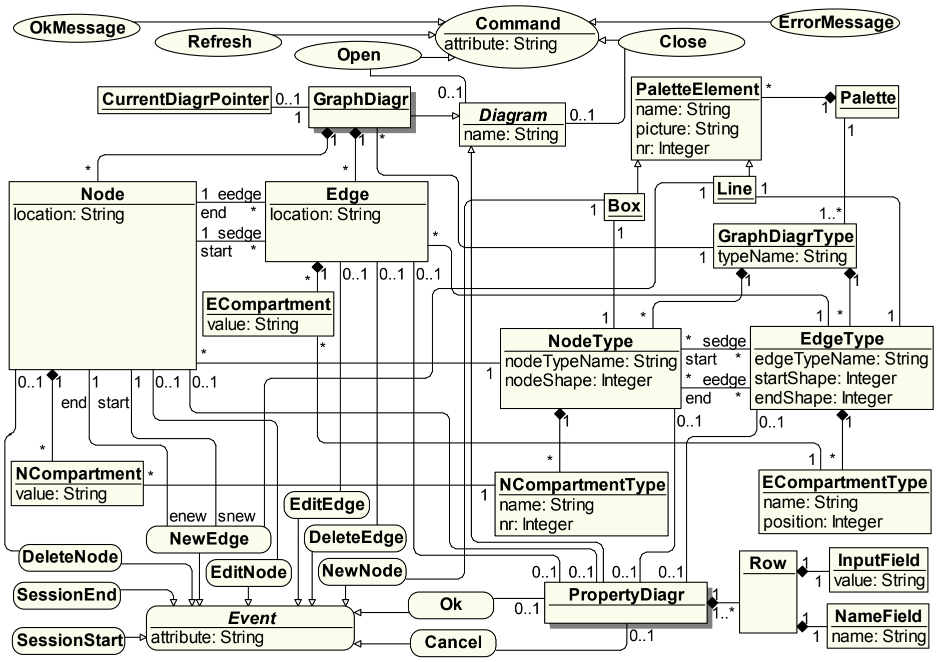

Fig. 2. Simplified interface meta-model

The Transformation Engine may, upon transferring the control back to PE, create *commands* (⬭, instances of *Command* class) to be executed by PE. If a command is of *Refresh* class, it requires the PE to update the visual presentation in accordance to changes that have occurred in the interface model. This type of command is typically used after node or edge addition or deletion in the visual presentation (interface).

If, for instance, a *NewNode* event is created in a simple flowchart diagram editor, the transformation finds the type of the node to be created via the link to a *Box* (*PaletteElement*) and then via the link to *NodeType*. If, for instance, the node type has *nodeTypeName* = “StartNode”, the transformation may check, if there is already an instance of class *StartNode* in the flowchart diagram, and if there is not, the transformation may insert a *Node* instance in IM and a *StartNode* instance in DM, and construct a scaffolding link between these instances. Then a command of *Refresh* type is created and control is returned to PE. If the transformation is not successful, an *ErrorMessage* command is created, what PE uses further to display a corresponding error message to the user. The transformations can also instruct PE to open property diagram, e.g. for input of the node attribute values.

An important design issue concerning IMM is that of the event and command granularity: what kind of user actions are served just by GrTP engine (is hard-coded into it), and what kind of work is delegated to transformations. The hard-coded part of functionality allows for transformations not to be overwhelmed by low level user interface manipulation, however, this kind of decision will restrict the possibility to adjust the behavior of the engine to the needs of a concrete specialization. For instance, in GrTP a design decision has been made to leave all work of positioning elements in the diagram to the hard-coded part. Another decision has been to allow the engine itself to produce the events like “New node” and “New edge”.

The full interface meta-model in addition to the simplified version considered here contains: *child-parent* association between nodes for representing node inclusion into other nodes, *subType-baseType* association between element type classes for implementation of stereotype-like mechanism, additional graphical element *Port* that is attached to the node and can be used as Pin representation in activity diagram. The meta-model also contains additional interface constructs along with corresponding events and commands, e.g., *Toolbar*, *PopUpDiagram* as well as transformation-driven menu for property diagram.

4.IMPLEMENTATION AND FIRST EXPERIENCE

The visual presentation part of GrTP is developed on the basis of graphical engines that are developed for GRADE tools family [6] and offer advanced graph drawing capabilities [4, 10], e.g., advanced services for automatic diagram element positioning according to different layouts, basic graph diagram editing services (e.g. Zoom, Print, Scroll, etc.).

To ensure the efficiency of GrTP, a novel model transformation language L0 has been implemented [12] with a highly efficient compiler; also an efficient in-memory repository for holding domain and interface meta-models and their instances has been developed.

The first experiments with GrTP have been performed in parallel with tuning up the platform itself. As a first implementation on the basis of GrTP, a class diagram editor for MOF-type meta-model building was created. As an example, the meta-models in this article have been constructed and printed using this meta-model editor. The time for transformation writing for this editor can be estimated in 3 person-months; that is several times less than creating a corresponding tool from scratch.

On the basis of our platform we have also developed an UML activity diagram editor covering nearly all their features, including elements like Pin, Interrupt region, etc., that are rather complicated from diagram editing viewpoint. This editor also has achieved a completely satisfactory performance level. The time for transformation writing for this editor can be estimated in 2 person-months. We expect that use of higher level model transformation languages could reduce the transformation writing time for new specializations of GrTP approximately by the factor of 2.

5.CONCLUSIONS AND FUTURE WORK

Our basic conclusion is that such tool building platform is sufficiently effective from practical point of view, both from perspective of efforts involved in specialization of GrTP into concrete tools (by means of transformation writing), and the performance of obtained concrete tools.

We have considered in this paper only interface meta-model for graph diagrams (together with some auxiliary diagrams), thus describing a graph diagram engine. We have under development also a tree diagram engine that is relatively simpler and can easily be integrated into the current architecture. The architecture admits also further inclusion of engines for other diagram types that use the event/command communication pattern with the transformation engine.

Our current plans include further development of graph diagram engine to enable it to serve animation of diagrams.

We see one of further target application areas for GrTP platform also in the context of Semantic Web. GrTP seems very suitable for building graphical editors for Ontology languages (RDF, OWL), and graphical Semantic Web browsers. One of the strengths of GrTP that can be exploited here is its ability to process large diagrams, both in the sense of performance, and in the sense of automated graph drawing facilities that are built into the engine. In this context it is important to stress the possibility to connect GrTP to other in-memory data stores (for instance Sesame [16]), which are more popular in Semantic Web world.

6.REFERENCES

[1] S. Cook, G. Jones, S. Kent and A. C. Wills, “Domain-Specific Development with Visual Studio DSL Tools,” Addison-Wesley, 2007.

[2] DiaGen Overview, http://www2.cs.unibw.de/tools/DiaGen/doc/Overview.pdf, 2000.

[3] DiaGen, The Tree Tutorial, http://www2.cs.unibw.de/tools/DiaGen/doc/Tutorial.pdf, 2003.

[4] K. Freivalds and P. Kikusts, “Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing,” Proc. The Latvian Academy of Sciences, Section B, Vol. 55, No. 1, pp. 43–51, 2001.

[5] The Generic Modeling Environment, GME User's Manual http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf, 2005.

[6] GRADE tools, http://www.gradetools.com/, 2007.

[7] Graphical Modeling Framework (GMF, Eclipse Modeling subproject), http://www.eclipse.org/gmf/, 2007.

[8] J. Grundy, J. Hosking, N. Zhu1 and N. Liu1, “Generating Domain-Specific Visual Language Editors from High-level Tool Specifications,” 21st IEEE International Conference on Automated Software Engineering (ASE'06), pp. 25-36, 2006.

[9] A. Kalnins, J. Barzdins and E. Celms, “Model Transformation Language MOLA,” Proc. MDAFA 2004, Vol. 3599, Springer LNCS, pp. 62-76, 2005.

[10] P. Kikusts and P. Rucevskis, “Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors,” Proc. Graph Drawing ’95, Lecture Notes in Computer Science, vol. 1027, pp. 361–364, 1996.

[11] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle and P. Volgyesi, “The Generic Modeling Environment,” Workshop on Intelligent Signal Processing, pp. 6, 2001.

[12] Lx Transformation Language Set, http://Lx.mii.lu.lv/, 2007.

[13] MetaEdit+ Method Workbench User’s Guide, Version 4.0, http://www.metacase.com/support/40/manuals/mwb40sr2a4.pdf, 2005.

[14] MetaEdit+ User’s Guide, Version 4.0, http://www.metacase.com/support/40/manuals/mep40sr2a4.pdf, 2005.

[15] MOF QVT specification, http://www.omg.org/docs/ptc/05-11-01.pdf, 2005.

[16] Sesame, http://www.openrdf.org, 2007.

[17] A. Shatalin and A. Tikhomirov, “Graphical Modeling Framework Architecture Overview,” Eclipse Modeling Symposium, 2006.

[18] N. Zhu1, J. Grundy and J. Hosking, “Pounamu: a meta-tool for multi-view visual language environment construction,” Proc. IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC’04), pp. 254-256, 2004.