The Transformation-Driven Architecture

Janis Barzdins   Sergejs Kozlovics   Edgars Rencis

Institute of Mathematics and Computer Science, University of Latvia
Raina blvd. 29, LV-1459, Riga, Latvia
{janis.barzdins, sergejs.kozlovics, edgars.rencis}@lumii.lv

Abstract

This paper proposes a new system building (in particular, tool building) approach, which we call *Transformation-Driven Architecture* (TDA). The basic elements used in this architecture are model transformations, interface metamodels with corresponding engines, and event/command mechanism. The implementation of the UNDO functionality in TDA is also sketched in this paper.

Keywords model transformations, metamodels, events, commands, UNDO

1. Introduction

The increasing popularity of domain-specific languages influenced the appearance of metamodel-based tools such as MetaEdit+ [1], Eclipse GMF [2], AToM^3 [3], Microsoft DSL Tools [4], DiaGen/DiaMeta [5], Pounamu [6], Marama [7], METAclipse [8], and GrTP [9]. Usually, the interaction between the domain and the presentation model is the cornerstone of such tools. The link between these two models may be established by static-mapping approach or by model transformations. However, presentation model is not yet the end product that can be shown to the user. This model should be processed by some engine and displayed in the appropriate form (for instance, as a diagram) to the user. Thus, we get the structure shown in Fig. 1.

According to the Model-View-Controller (MVC) [10] architectural pattern, the domain can be viewed as a model, the presentation — as a view, and the processes that map them — as a controller. If the model and the presentation do not communicate directly, but by means of model transformations, the analogy with the Three-tier-architecture [11] can be noticed. The model corresponds to the data tier, the presentation — to the presentation tier, and the transformations — to the application (or logic) tier. Some other analogies with the Three-tier architecture may also be noticed (see triple-lines in Fig. 1).

In this paper we will go further. The structure from Fig. 1 can be extended by adding additional presentation metamodels and the corresponding engines. The logic that links the domain with all the presentations can be implemented by means of model transformations. In this way a new system building approach arises, which we call *Transformation-Driven Architecture* (TDA).

The following differences between TDA and the approaches used in the abovementioned tools could be realized. Most of the tools have only one presentation that is handled globally by the platform. TDA, in its turn, allows using several different presentations, and each of them can be handled independently by the corresponding engine. Another difference is the wide usage of model transformations in TDA. Transformations can be used not only for linking the domain to one or more presentations, but also for exchanging data between different presentations.

TDA allows transformations to communicate with engines by means of commands and events. Commands are used when engines are called by transformations, while events are used when transformations are called by engines. This differs from the approach used in METAclipse [8], where only events[1] are used: when some presentation event occurs, the platform creates the corresponding object X in the repository and calls the transformation. The transformation may return certain kind of information back to the platform through the same object X. In this approach the transformation is not able to access the platform's functionality directly: it has to use the object X. TDA doesn't have this limitation since both events and commands are used. Not only engines may call transformations, but also transformations are allowed to call engines, which, in their turn, may call other transformations, and so on.

The paper is structured as follows. The next section lists some assumptions setting the background for the explanation of the TDA. Sect. 3 depicts the idea of the TDA and explains the collaboration between transformations and engines. Sect. 4 sketches the solution for implementing the UNDO functionality being a certain issue, which is not implemented (or implemented by storing/loading the model, which is slow) in some tools. In TDA this problem becomes more complicated since not only transformations but also engines have to undo their actions. Although we don't present the whole solution here, we show that the UNDO implementation is possible in TDA. Finally, Sect. 5 concludes this paper.

2. Technical Assumptions for TDA

In this section we list some technical assumptions setting the background for the TDA. The assumptions are as follows.

- The data is stored in some repository (like EMF [12], JGraLab [13] or Sesame [14]) with fixed API (Application Programming Interface).
  Motivation. Fixed API simplifies the way of accessing the repository by engines or by transformations. Fixed API is needed also for UNDO implementation.
- The API of the repository should be available for one or more high-level programming languages (such as C++ or Java), in which presentation engines will be written.
  Motivation. This allows the engines to be able to exchange the data with transformations through the repository.
- Model transformations may be written in any language (for instance, any textual language from the Lx family [15] or the graphical language MOLA may be used [16]). However, the transformation compiler/interpreter should use the same repository API as the engines.
  Motivation. This will simplify the UNDO implementation.
- When a transformation is called, its behavior depends only on the data stored in the repository.
  Motivation. This will also simplify the UNDO implementation.

[1] They are called "commands" in METAclipse, while in our context they play the role of events.

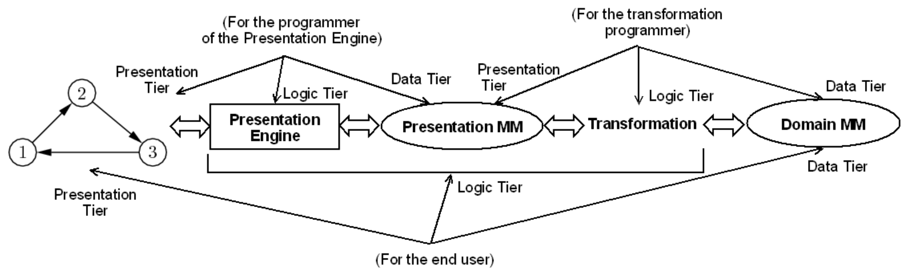

Figure 1. The chain between the end user presentation and the domain. Triple-lines show relations to the three-tier architecture.

If the transformations needs some internal variables, these variables may be stored in the repository as well.

- The metamodels do not change at runtime. All the changes in the repository are at the instance-level.
  Motivation. This is a simplification only. The changes at the metamodel-level can be considered in the same way as changes at the instance-level, if needed.
- Only one repository is used. Interaction between different repositories as well as distributed data storage are not considered.
  Motivation. This is also a simplification. A transparent super-API over different/distributed repositories may be defined, if needed.
- Only one module (transformation or engine) is allowed to access the repository at the same time. Concurrency and locking issues are not considered.
  Motivation. On the one hand, concurrency and locking issues are complex enough and require separate research. On the other hand, most metamodel-based tools don't have concurrent access to the repository, and, usually, it is not so essential.

3. The Transformation-Driven Architecture

Fig. 2 shows a sample system that depicts the essence of the TDA. The cloud in the center is the system metamodel that may be divided into several parts. In our example, one can notice parts called "Domain metamodel" and "Presentation metamodel". The presentation metamodel has the corresponding engine, and, in fact, the whole chain from Fig. 1 can be found also in Fig. 2. There are also other presentation metamodels (the dialog metamodel, the database metamodel and the XML metamodel in our example) with their engines. Each presentation metamodel can be considered as an *interface metamodel* for the corresponding engine.

Let's call the remaining part of the system metamodel (to which the class "Command" belongs) the core part. The five parts mentioned above are linked to this core part, thus complementing it. The core part also has its own engine called "Head engine". The head engine may be considered as an "operating system" (OS), and the other engines as device drivers. The real functionality is performed by model transformations that may be considered as programs.

3.1 How Does the System Work

In the beginning the control is gained by the head engine. It constructs the main window, where the user can open an existing project or create a new one. In the first case the data is simply loaded to the repository. In the second case the head engine calls the corresponding transformation that fills the repository with some initial data.

When the project has been initialized, each engine may generate events, and the corresponding transformation being able to handle this event is executed. Before calling the transformation, the engine in which the event occurs creates an instance of the corresponding "Event" subclass. The properties (attributes and links) of this instance may be considered as arguments for the transformation.

While events are used to call transformations from engines, commands are used for the opposite direction — to call engines from transformations. When there is a need to call an engine, the transformation creates a command and asks the head engine to execute this command like a program may call an OS function to get access to some device. The head engine determines which of the engines must be called and passes the control to it. That is like the OS passes the control to the corresponding device driver.

Each command is an instance of some "Command" subclass. For example, a command for the Presentation Engine may be of the type "PresentationCommand". The presentation engine may have several command types that are descendants of the class "Command" and that belong to the presentation metamodel, so the appropriate type must be selected and the instance created (see also [9]).

In order to execute the given command, the head engine checks the type of the corresponding instance and determines the part of the metamodel to which it belongs. After that the corresponding engine is called.

While a command is being executed, events may be created and transformations may be called. These transformations may also create commands. While existing commands are being executed, new commands have to be managed correctly. Thus, the appropriate data structure is required to store the commands. We call it the command queue, however, in reality it is a hybrid of the queue and the stack.

3.2 The Command Queue

Assume that during execution of some command other commands are not created. In this case the command queue is a real queue. Let there be a fictive command EOC ("End of Commands") denoting the end of the command queue (EOC is the command after the last real command). We assume there is a pointer to this EOC instance. When a transformation needs to add a command, it finds the EOC and inserts the new command just before the EOC by creating the corresponding "previous"/"next" links. When the head engine processes the queue, it finds the first command and executes the commands starting from the first until the EOC is reached.

However, if execution of a command creates other commands, the new commands should not be added to the end of the queue. Fig. 3(a) depicts the point. Assume that during the execution of command A the transformation, which adds additional commands A_1 and A_2, is called. A is still being executed. So, A_1 and A_2 should be considered as parts of A. Thus, A_1 and A_2 have to be executed before B, not after C as it would be if A_1 and A_2 were added just before the EOC.

In order to solve this problem, we modify the command queue slightly. Before starting to execute A, it is replaced by EOC, and

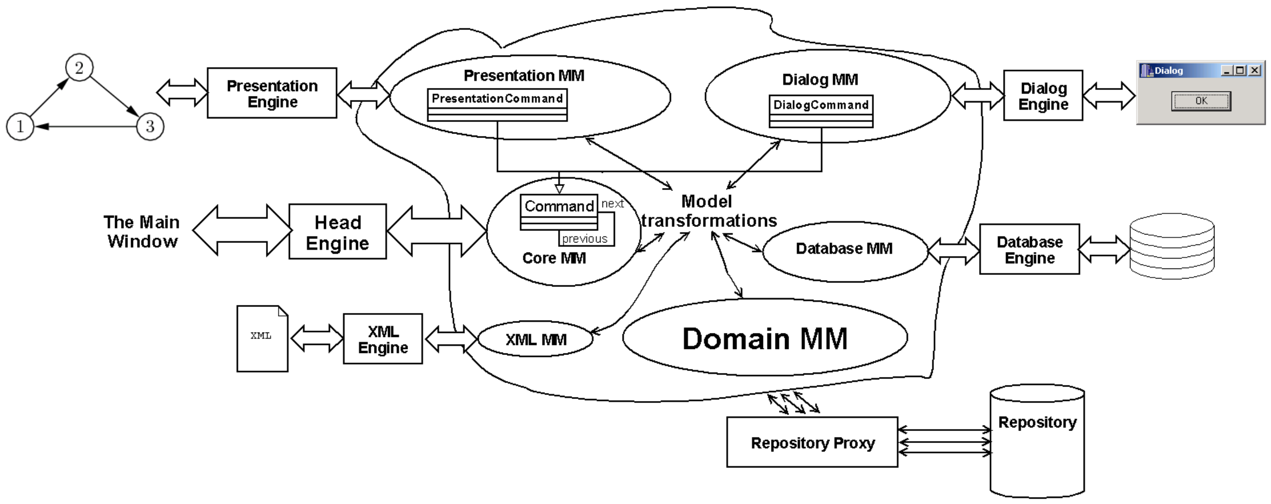

Figure 2. An example revealing the essence of the Transformation-Driven Architecture.

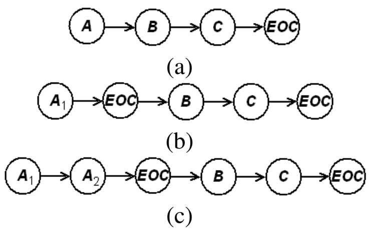

Figure 3. (a) A sample command queue. (b)–(c) While processing command A two more commands are added — A_1 and A_2.

the EOC pointer is moved to this new EOC. Now, we can execute A. If new commands need to be added, they are added before the just created EOC, and, thus, also before B (see Fig. 3(b) and (c)).

So, the command queue is actually a stack of queues, where the queues are separated by EOCs (see Fig. 4).

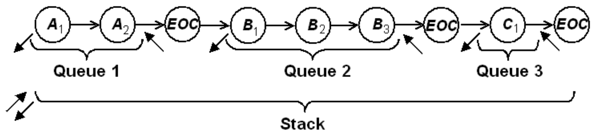

Figure 4. The structure of the command queue (the "stack of queues").

4. The Implementation of UNDO

An important question arises whether it is possible to implement the UNDO functionality in TDA. It is worth to note that during the UNDO not only a) the actions performed by transformations have to be undone, but also b) the engines must be returned to the corresponding states.

The answer to the question just mentioned is "Yes". The problem a) can be solved by accessing the repository through the proxy that manages the UNDO history. The problem b) is solved by introducing additional functionality that can be used to notify the proxy when the engines change their states. We sketch these solutions below.

In order to solve the problem a), some repository API functions are hooked by the repository proxy. That is why the API has to be fixed.

The repository API contains read-only functions (such as functions for traversing the repository) as well as functions that modify the repository (modificating functions). Modificating functions can be divided into primitive and non-primitive ones. We assume the following instance-level functions to be primitive:

- adding/deleting an instance;
- adding/deleting a link between two instances;
- changing the value of some attribute (property).

(We could select also certain metamodel-level routines if the metamodel could be changed at runtime.) It is considered that non-primitive modificating functions can be implemented by means of primitive functions. For instance, the function for cascade delete (that deletes the aggregate with its parts recursively) can be implemented through calls to the function for deleting single instance and to the function for deleting single link. This allows to handle all repository changes by considering only primitive functions. Since engines and transformations use the same API, hooking primitive functions handles all the changes made in the repository.

When a primitive function is called, the information needed to undo and also to redo the corresponding operation is being written into the history. The history consists of *transactions*. Each transaction contains undo/redo information for several actions. Transactions are started before certain events like "New box" or "New line" events in the presentation engine.

The solution for the problem b) involves notifying the proxy about the changes in the states of one or more engines. Each engine may have several units for which UNDO can be called independently. We call these units *diagrams*. For instance, there may be several open graph diagrams managed by the Presentation Engine. If the user had changed Diagram A before Diagram B, then UNDO in Diagram A should not affect Diagram B (unless the diagrams depend on each other). For each diagram the engine has to know how to track the changes and to undo them. For instance, in graph diagrams the coordinates of objects may be saved, and, when UNDO is called, restored. The issues concerning what the diagram is and how its states are saved and restored depend on the engine.

For UNDO we are interested not in engines, but in diagrams that have been changed (since during the same transaction some diagrams of the same engine may be changed, and some may be left unchanged). The function `DiagramChanged(DiagramID, CurrentStateID)` is used to notify the proxy about diagram changes. The diagram is being changed from the previous state to the state `CurrentStateID`. `DiagramChanged` should also be called for each diagram to specify initial states in the beginning. It is assumed that the diagram change corresponds to the current transaction. Thus, undoing this transaction will return the diagram to the state before the change.

The list of other UNDO-related functions is presented below (but without implementation details):

- `CreateCheckPoint()` — creates a new transaction; should be called before certain events.
- `CanUndo(DiagramID)` — returns `true` iff the last transaction that modified the given diagram can be undone.
- `Undo(DiagramID)` — undoes the last transaction where the given diagram has been modified (let's call this transaction T_1). Other transactions may also be undone. For instance, if another diagram has been modified in T_1 and also in transaction T_2, where T_2 comes after T_1, then T_2 also needs to be undone. Thus, other diagrams may need to change their states as well. For that reason, `Undo` returns the list of diagrams and the corresponding states.
- `CanRedo(DiagramID)` — returns `true` iff there was an undone transaction where the given diagram has been changed. This transaction can be redone.
- `Redo(DiagramID)` — acts similar to `Undo`, but in the opposite direction.
- `ClearHistory(DiagramID)` — deletes from the history all the transactions that affected the given diagram. Some other transactions may also be deleted to keep the history consistent.
- `ClearAllHistory()` — clears all UNDO history.

In fact, functions for creating dependencies between transactions may also be introduced. For instance, the function `ObjectMustExist(ObjectID)` may be used to indicate that undoing the transaction where the given object has been created forces undoing also the current transaction.

5. Conclusion

The main contribution of the paper is the idea of using several presentation metamodels with the corresponding engines, where the connection between all the metamodels (including the domain metamodel) is ensured by model transformations. The command queue has been introduced as the way of transferring control between transformations and engines. The basic ideas of the UNDO implementation have also been presented with the aim to show that such an implementation is possible. A detailed explanation and comparison to other approaches for UNDO are subject to further research.

In this paper, we haven't addressed the problem of writing model transformations to be used with TDA. One of the solutions is to write transformations from scratch each time a new system is being built. Another solution is to write a universal transformation, which would handle many typical tasks arising in the tool building process. A special tool definition metamodel may be introduced, and the universal transformation may interpret instances of this metamodel. This research topic is also of our interest.

Some ideas presented in this paper have been successfully implemented in the recent version of transformation-based tool building platform *GrTP* [9]. There are two presentation engines (the graph diagram engine and the dialog engine), which communicate with model transformations by means of commands and events. Commands are stored in the command queue, and there are cases when the command queue has to be modified by introducing EOC command (for instance, when one modal dialog calls another). The transformation-driven architecture has proved to be powerful enough to build an experimental tool — an editor for UML class diagrams with profiles (stereotypes) and other advanced features. Moreover, a graphical query tool for RDF databases has also been built using this architecture [17].

It is a well-known fact that the development of presentation engines is a very time-consuming process. That's why it is done only once in our approach. On the contrary, it is relatively easy to write transformations working with metamodels. And this is the basic profit we can achieve by means of TDA.

Acknowledgments

The authors would like to thank Karlis Cerans, Renars Liepins and Sergejs Rikacovs for their help and suggestions.

References

[1] MetaEdit+, `http://www.metacase.com`

[2] A. Shatalin and A. Tikhomirov. Graphical Modeling Framework Architecture Overview. *Eclipse Modeling Symposium*, 2006.

[3] J. de Lara, H. Vangheluwe and M. Alfonseca. Meta-Modeling and Graph Grammars for Multi-Paradigm Modeling in AToM3. *Software and System Modeling*, 3(3), 2004., pp. 194-209.

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

[5] DiaGen/DiaMeta. `http://www.unibw.de/inf2/DiaGen/`

[6] 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), 2004., pp. 254-256.

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

[8] A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks and J. Barzdins. Building Tools by Model Transformations in Eclipse. *Proceedings of DSM'07 Workshop of OOPSLA 2007*, Montreal, Canada, Jyvaskyla University Printing House, pp. 194-207.

[9] J. Barzdins et al. GrTP: Transformation Based Graphical Tool Building Platform. *Proceedings of MDDAUI'07 Workshop of MODELS 2007*, Nashville, Tennessee, USA.

[10] T. Reenskaug. The Model-View-Controller (MVC). Its Past and Present. *JavaZONE*, Oslo, 2003. JAOO, Arhus.

[11] Wayne W. Eckerson. Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications. *Open Information Systems 10*, 1 (January 1995): 3(20).

[12] Eclipse Modeling Framework (EMF, Eclipse Modeling subproject), `http://www.eclipse.org/emf`.

[13] S. Kahle. *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*, Diplomarbeit, University of Koblenz-Landau, Institute for Software Technology, 2006.

[14] Sesame. `http://www.openrdf.org`.

[15] J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs. Model Transformation Languages and their Implementation by Bootstrapping Method. *Pillars of Computer Science*, Springer LNCS, Vol. 4800, 2008., pp. 130–145.

[16] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA, *Proceedings of MDAFA 2004* (Model-Driven Architecture: Foundations and Applications 2004), Linkoeping, Sweden, pp. 14–28.

[17] G. Barzdins, E. Liepins, M. Veilande, M. Zviedris. Semantic Latvia Approach in the Medical Domain. *Proceedings of the 8th International Baltic Conference* (Baltic DB&IS 2008), Tallin, Estonia, pp. 89–102.