ISSN 0146-4116, Automatic Control and Computer Sciences, 2013, Vol. 47, No. 1, pp. 28–37. © Allerton Press, Inc., 2013.
Original Russian Text © S. Kozlovics, J. Barzdins, 2013, published in Avtomatika i Vychislitel'naya Tekhnika, 2013, No. 1, pp. 39–52.

The Transformation-Driven Architecture for Interactive Systems

S. Kozlovics[a, b] and J. Barzdins[a, b]

[a] *Institute of Mathematics and Computer Science, University of Latvia, bulv. Raina 29, Riga, LV-1459 Latvia*
e-mail: sergejs.kozlovics@lumii.lv
[b] *University of Latvia, bulv. Raina 19, Riga, LV-1586 Latvia*
e-mail: janis.barzdins@lumii.lv

Received October 9, 2012; in final form, November 28, 2012

Abstract—We propose new software architecture for interactive systems (systems with interacting components). This architecture, called the Transformation-Driven Architecture, TDA, uses certain ideas of the Model-Driven Architecture, MDA. However, unlike MDA, which uses models and model transformations at software development time, TDA uses them at runtime. To describe the dynamics of the system as well as interaction between the components, TDA uses special objects called events and commands.

Keywords: Transformation-Driven Architecture (TDA), Model-Driven Architecture (MDA), interactive systems

DOI: 10.3103/S0146411613010057

1. INTRODUCTION

1.1. The Model-Driven Architecture

Different methods exist for automatiic creation of computer programs, including the application of high-level programmming languages and factoring out recurring subproblems (e.g., creating function libraries or class libraries). One of the advanced methods is using models and their transformations in the course of software development. The Model-Driven Architecture (MDA) can serve as the starting point for this approach [1, 2]. It was proposed by the Object Management Group (OMG) in 2001.

In MDA, the model is taken to mean the description of a system in a language that has clearly defined syntax and semantics to allow this language to be automatically interpreted by a computer [1, p. 16]. Since such a language is used to describe models, it is a model itself, but at another level. This language (the model) describes other models; therefore, it is called a metamodel. If a model M is described by a metamodel MM, then M is said to conform to MM.

A special language is required to desrcibe metamodels. The Meta-Object Facility (MOF) standard that has been developed by the OMG determines two variants of this language, namely, Complete MOF (CMOF) and Essential MOF (EMOF) with the latter being a simplified version of CMOF [3]. In practice, the ECore language, part of the Eclipse Modeling Framework, is used, which is very similar to EMOF [4].

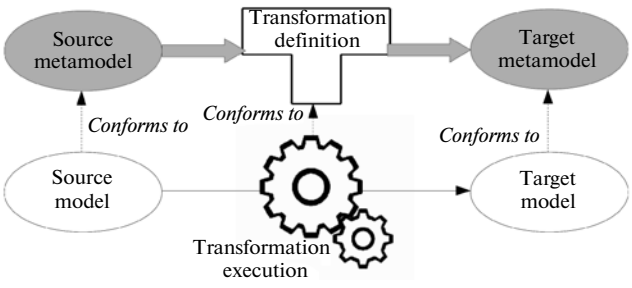

Fig. 1. Definition and execution of a model-to-model transformation.

The model-to-model transformation is the process of data transformation where input and output data are models conforming to certain metamodels. Transformations are defined for metamodels but they are executed on models (Fig. 1). Also, model-to-something transformations (commonly, from model to text) and something-to-model transformations (commonly, from text to model) exist. Specialized languages such as transformaton languages and mapping languages exist for defining transformations.

The essence of MDA is as follows. Originally, the program is described as a model rather than a code at a sufficiently high abstraction level using

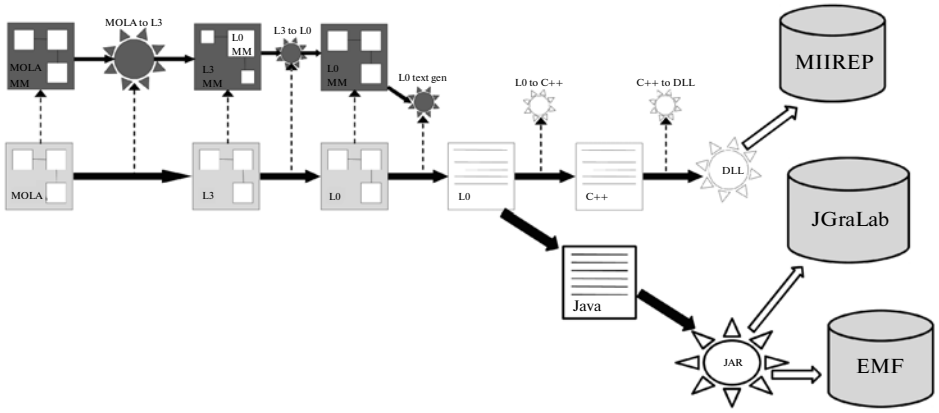

Fig. 2. Of the MOLA language compiler. The process of compilation of MOLA programs corresponds to the MDA principles (with the kindest permission of the author—Agris Šostaks).

the concepts from a given domain. This model is called the computation-independent model (CIM). Then, this model is transformed to the platform-independent model (PIM), which does not depend on specific technologies (e.g., .NET or Java)[1]. Therefore, when new technologies appear, CIM and PIM stay unchanged, which allow one to maintain investements in these models. PIM, in turn, is transformed to a platform specific model (PSM), which is subsequently transformed to code in some programming language. In certain cases, the process starts from PIM rather than CIM. Moreover, several intermediate PIM/PSM models are possible, each of which plays the role of either PIM or PSM depending on whether the model is the source or the target one.

Therefore, models in MDA are more similar to the source code rather than the diagrams drawn on paper. Using them and a number of transformations, a ready program is created. Such models are valuable artifacts during the entire lifetime of the program. Ideally, all model transformations should be executed automatically. Actually, only part of the transformations is fully automated.

1.2. Advantages and Disadvantages of MDA

MDA is an intrinsic step in using models in software production. The MOF standard and the derivative ECore standard are commonly admitted metalanguages [3, 4]. Moreover, OMG has given impetus to the emergence of model transformation languages [6]. MDA is also the starting point in a more extensive research domain called Model-Driven Engineering (MDE). MDA is really applicable to practical cases. By way of example, the MDA principles are used when compiling transformations written in the MOLA language [7, 8]. The source platform-independent model in the MOLA language (PIM) is transformed to the chain of intermediate PIM/PSM models. Finally, code in the *C++* or Java languages is generated, which is capable of working with one or another model repository (Fig. 2). However, MDA in its pure form is not applicable everywhere. Today, MDA is subjected to criticism [9–12]. In line with D. Thomas, we believe that the generation of more or less complex PSMs (e.g., in order to describe the entire .NET environment), as well as the respective model transformations, would require supreme efforts [11]. Such efforts will also be required afterwards when the platform changes or some new platforms appear. In addition, MDA does not explain how the dynamics of the behavior of the system under development should be described, e.g., the reaction to certain events. Moreover, if the system uses external modules, then the interaction of the system with these modules should be provided. We can describe such dynamics in the model itself (PIM), but, in this case, the description is essentially a program that is coded as a model, which makes the model complicated and nullifies all the advantages of MDA.

The authors of this paper, based on some ideas of MDA, are proposing a software architecture for interactive systems. By an interactive system we mean a software system with interacting components. Such systems require a method for describing dynamics and a certain mechanism for communication between

[1] The scientific school headed by professor J. Osis proposed an interesting approach to CIM description based on which one can automatically obtain PIM and then a ready system [5]. Its feature is that everything that is needed is described in CIM, including the functional cycles according to which the system would operate.

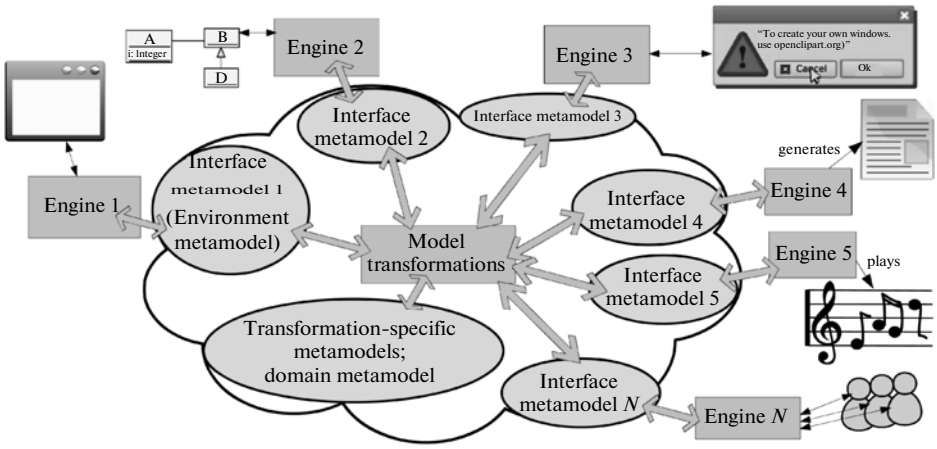

Fig. 3. The transformation-Driven Architecture (TDA): a schematic view.

the system's components. The proposed architecture, which is called the Transformation-Driven Architecture (TDA), intends to resolve these problems.

Unlike MDA, where models and transformations are used at development time, TDA uses them at runtime. However, TDA with its models and transformations can also be used at development time.

2. THE TRANSFORMATION-DRIVEN ARCHITECTURE

TDA is scematically represented in Fig. 3. Sections 2.1–2.4 describe engines, metamodels, and model transformations. We will describe how TDA provides the interaction of engines and transformations in sections 2.5–2.6.

2.1. Engines

When developing a system based on the TDA principles, we should primarily determine what additional functionality is required to implement the main functionality (business logic) of the system. Additional functionality is implemented by the modules that are called engines. The engines are responsible for the following:

- Data representation to user (e.g., in the form of graph diagrams or dialog windows).
- General-purpose services. These services can be responsible for, e.g., the generation of Word documents, sending Email messages, printing out data, using multimedia opportunities, generating XML documents or programs according to specified rules, etc.

Engines are commonly implemented in traditional programming languages, which are the most convenient for implementing the additional functionality (different engines can be written in different languages). The implementation of engines is often platform-specific since it can depend on the functions of a particular platform (e.g., Java or NET) or a library that is written in a particular programming language and is dependent on a particular compiler/interpreter. In certain cases, a functional or declarative programming language may be advisable for implementing an engine.

2.2. Interface Metamodels

In terms of the main functionality (business logic), the internal implementation details of an engine are not essential. In TDA, the implementation of business logic requires the description of only the essential details of each engine. Engine essences, or interfaces, are described by means of metamodels that are called interface metamodels. Each engine must be capable of working with the models conforming to its interface metamodel and providing a link between these models and the implementation of its functionality.

Interface metamodels are analogs of metamodels for describing platform-independent models (PIMs) in MDA. However, in contrast to MDA, TDA uses several PIMs rather than one, which makes it possible to share the problem among several engines. In addition, in TDA, the engines interpret models, i.e., they

set links between the PIM model and the platform-dependent implementation of functionality (in MDA, PIM is transformed to PSM and then to the platform-dependent code at development time).

2.3. Model Transformations

Model transformations (in the center of Fig. 3) implement the main functionality (business logic) of the system. If engines are comparable to model-to-something or something-to-model transformations, then business logic is described by model-to-model transformations. The main distinctions of TDA transformations from MDA transformations are as follows:

• TDA transformations are commonly incremental; i.e., they are allowed to change existing models. Incremental transformations are convenient for implementing interactivity (see sections 2.5–2.6).

• Each TDA transformation can work with any number of models (even with a single model); some of the involved models will be used as the source, while others, as the target. Some will be used as source and target ones at the same time.

• Transformations are launched at runtime.

Transformations can be described using:

• Traditional programming languages.

• Special-purpose transformation languages such as ATL [13], the Epsilon family of languages [14], MOLA [7], GReAT [15], GreTL [16], etc...

• Mapping languages, e.g., MALA4MDSD [17], which can describe (commonly in a declarative manner) how a target model can be derived from a source model. Mapping languages can be considered as high-level transformation languages.

• Web ontology language (OWL) constructions [18, 19]. Although OWL describes ontologies rather than transformations, the process of launching the semantic reasoner for a given ontology can be compared to the transformation that supplements a model with inferred data.

2.4. Other Metamodels and Types

In TDA, transformations can work not only with the interface models of engines. Transformations can introduce their own models (and corresponding metamodels for their description) that are required for implementing the business logic. One of the examples is the domain model that is the central model for storing business logic data using the given domain concepts. Domain models are extensively used in Domain-Specific Modeling (DSM) [20].

All the metamodels and models[2] conforming to them in TDA are stored in a repository.

2.5. Interactivity in TDA

The following approach is used in TDA to suport the interactivity (interaction of engines and transformations). Special classes are defined in the interface metamodels of engines that, depending on the direction of the interaction (from transformation to engine or vice versa), are called commands (the subclasses of the predefined Command class) or events (the subclasses of the predefined Event class). The command and event parameters can be coded by attributes or links to other parts of the metamodel (these links specify the context). This approach for describing the interaction is superior to using operations in the MOF/ECore style or using methods found in object-oriented programming for the following reasons:

• Not all types of repositories support operations in the MOF/ECore style.

• Operations describe only one-way calls; the description of callbacks is commonly related to additional technical complications.

When a transformation needs the functionality of an engine, it describes the problem in a model that conforms to the interface metamodel of the engine. Further, the transformation creates a command object and, if necessary, adds this object to the context. In order to launch the engine that implements the command, the transformation connects this command to a special submitter object. At this moment, TDA takes the control and calls the engine that implements the command. The command object is removed upon the command completion. The context objects can stay in the model if necessary.

In this approach, transformations use only operations with the model, and the introduction of additional constructions in the transformation languages is unnecessary for calling the handlers of commands

[2] Figure 3 shows only metamodels.

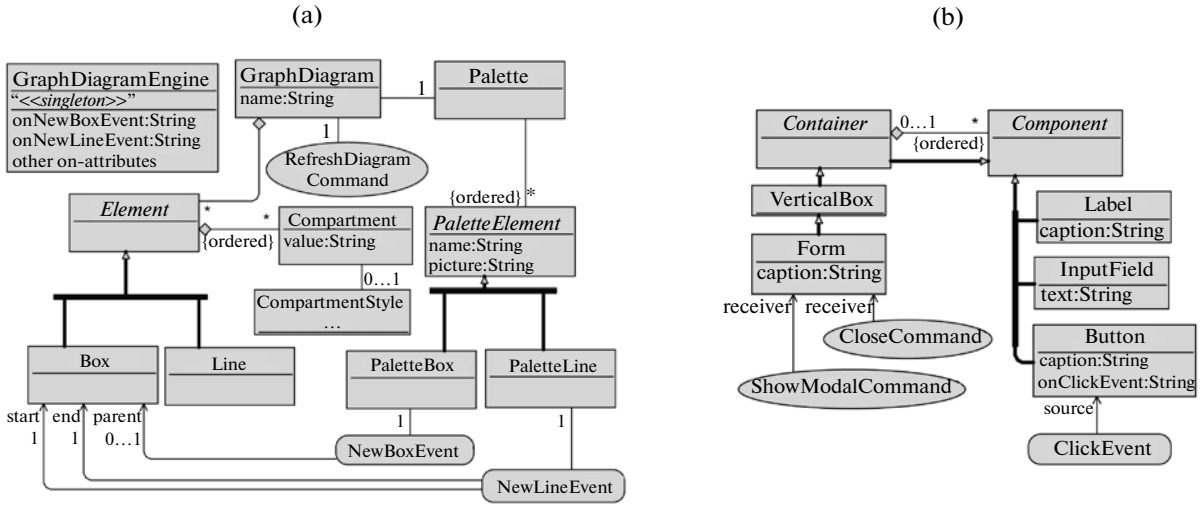

Fig. 4. (a) Fragment of a graph diagram engine metamodel. (b) Fragment of a dialog engine metamodel.

and events. In addition, there is no need for indicating which engine implements the command, since the engine is determined unambiguously for each command by its class, which is found in the interface metamodel of only one engine.

When the engine is active, it can inform the transformations on some events (e.g., the user has entered data or the engine has completed a certain part of the task). For this, the engine generates an event object and also links it with the submitter object. At this moment, TDA calls the transformation associated with this event. Upon event processing, the event object is deleted from the repository.

In order to associate transformations with events, special attributes called on-attributes are introduced in the classes that form the context of the event. The names of these attributes are created by adding the on prefix to the event names, e.g., onClickEvent. The value of the attribute is a string that contains the transformation name to be called when the given event occurs

2.6. An Example of the Interaction of Engines and Transformations in TDA

Let us consider a variant of the interaction of engines and transformations using a graphical tool for editing activity diagrams. This tool obviously requires the graph diagram engine [21] for displaying diagrams themeselves as well as the dialog engine [22] for inputting text information, e.g., the names of actions. Figure 4 shows fragments of the metamodels of these two engines.

Assume that the graphical tool has already been launched, and the diagram (represented in the model by the instance of the GraphDiagram class) has been opened. The diagram has already a pallete of graphical elements (represented by an instance of the Palette class and several instances of the PaletteElement class). Assume that the element (of PaletteBox type) that is responsible for adding a new action to the diagram is present in the palette.

When the user adds a new action to the diagram using the mouse, the graph diagram engine generates an event (an instance of the NewBoxEvent class) and links it to the corresponding instance from the PaletteBox and possibly to the instance from the Box class that indicates inside which box the user intends to add a new action. Once the graph diagram engine has linked the event object with the submitter object, TDA calls a transformation whose name is taken from the onNewBoxEvent attribute of the GraphDiagramEngine class.

When processing the NewBoxEvent, the transformation can create a dialog window wherein the user is allowed to input an action name. The transformation creates a dialog model in the repository where such components as static text (see the Label class), an input field (see the InputField class), and buttons (see the Button class) are placed in the dialog form (an instance of the Form class). For the OK and Cancel buttons, the values for the onClickEvent attribute are set to point to two other transformations. Depending on which button is pressed, either transformation will be called on the ClickEvent event.

The first transformation (the OK transformation) creates a box object (see the Box class) in the model of the graph diagram engine, adds a compartment in this box (see the Compartment class), and sets its

style (see the CompartmentStyle class) such as its text color, font size, etc. The OK transformation further calls the RefreshDiagramCommand command for refreshing the diagram with account for the new element and then closes the dialog window by means of CloseCommand. The dialog window can now be deleted from the repository.

The second transformation (the transformation for the Cancel button) creates nothing in the diagram and just closes the dialog window (and possibly deletes it from the repository).

2.7. The Use of TDA for Domain-Specific Tools

The aforementioned actions that describe the addition of a graphical element to the diagram will be similar in other tools designed for editing graph diagrams. These actions can be factored out and placed in the universal transformation library. Moreover, a special graphical metatool can be created for developing other graphical tools. If the universal transformation library (it is sufficient to develop it only once) is used in the tools being created, then these new tools can be created almost without (or totally without) writing transformations. This approach is implemented in the GRAF metatool that was developed at the Institute of Mathematics and Computer Science, University of Latvia [23]. GRAF itself, as well as the graphical tools being created using it, are based on TDA. They use the universal transformation library and the same engines.

GRAF was applied to create the GradeTwo editor of UML[3] diagrams (http://gradetwo.lumii.lv), the OWLGrEd tool for editing OWL ontologies (http://owlgred.lumii.lv), the ViziQuer tool for specifying graphical queries for semantic databases (http://viziquer.lumii.lv), and the ProMod and BiLingva tools for modeling business processes [24]. This indicates the viability of the proposed Transformation-Driven Architecture.

It is interesting that the transformations that are launched during the GRAF operation are used at runtime. However, in terms of the tools to be created, these transformations are used during development (as in MDA).

3. TECHNICAL DETAILS OF TDA

This section deals with additonal technical components of TDA and describes the procedure of launching and finishing TDA life cycle.

3.1. Technical View of TDA

Repository access interface. There exist different repository types for storing models, e.g., EMF Ecore [4], JR [25], and JGraLab[4]. A special abstraction layer called Repository Access API (RAAPI) [26][5] is introduced for operating on different repository types. This interface determines primitive operations on model elements, e.g., “create a class with a specified name” (CreateClass), “create an instance object of a specified class” (CreateObject), “create a link between two specified objects” (CreateLink), etc.

Adapters for repositories. A special adapter is created for each repository type to implement RAAPI and transform the RAAPI operation calls to the operation calls of the corresponding repository type (see Fig. 5; the repository adapters are designated by R).

TDA Kernel. Engines and transformations also use RAAPI to access the repository; however, the RAAPI calls are transferred to repository adapters through a special component called TDA Kernel (see the sphere in the center of Fig. 5). Therefore, the kernel can intercept and even modify RAAPI calls, which is useful for implementing certain functions.

TDA Kernel implements the following:

- Calling command and event handlers when the command object or the event object is linked to the submitter object.
- The undo/redo mechanism [27].
- The possiility of the simultaneous use of several repositories (probably of different types) by combining them into a single repository [26].
- Loading and storing the repository content.

[3] Unified Modeling Language
[4] http://www.ohloh.net/p/jgralab
[5] see http://tda.lumii.lv/raapi.html

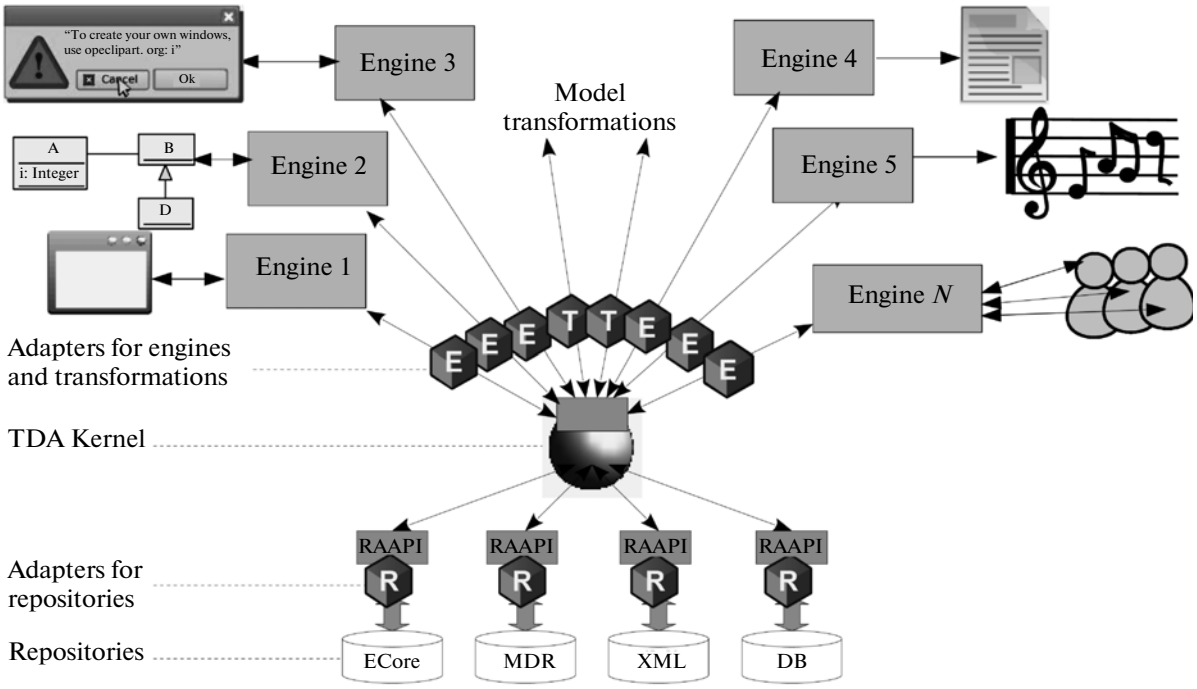

Fig. 5. Transformation-Driven Architecture (TDA): technical view.

TDA Kernel, similarly to the engines, also has its own interface metamodel and the corresponding model wherein the aforementioned classes Event, Command, and Submitter are defined, as well as other technical classes, e.g. the command for attaching an engine.

RAAPI wrappers. The current implementation of TDA Kernel (and RAAPI) is written in Java. In order to write engines and transformations in other programming languages (transformation languages) using the applicable technologies, wrappers are created for RAAPI, e.g., in the form of the native dynamic library (DLL) or the .NET library.

The implementation of some transformation languages such as MOLA, ATL, and Epsilon has certain abstraction layers for repository access [7, 13, 14]. Therefore, it is sufficient for these languages to implement the redirection of their abstraction layers to RAAPI.

Adapters for engines and transformations. This is important to provide a means to call engines and transformations, which can be written in different programming languages (transformation languages), in order to process commands and events in TDA. For this, adapters for engines and transformations are introduced in TDA (designated by E and T, respectively, in Fig. 5).

Each adapter has to implement an initialization function as well as a function for calling the corresponding engine or transformation.

During the initialization, the engines are able to create initial data in the model corresponding to their metamodel and the transformations are able to check the availability of all the required libraries. During the call, the engines search for command objects in the repository and the transformations search for event objects. Then, the actions concerning the processing of this command or the event are executed.

3.2. Launching and Completion of TDA Work.

The environment engine. One of the engines is distinguished particularly in TDA. It is responsible for the launching and finishing TDA work, as well as the creation of the main window wherein the windows of the other engines will be placed. This engine is called the environment engine.

We will describe briefly the metamodel of the environment engine (Fig. 6).

When the user intends to create a new project[6], open an existing one, or close an opened project (for this, the environment engine shows special options), then a corresponding event emerges viz. ProjectCreatedEvent,

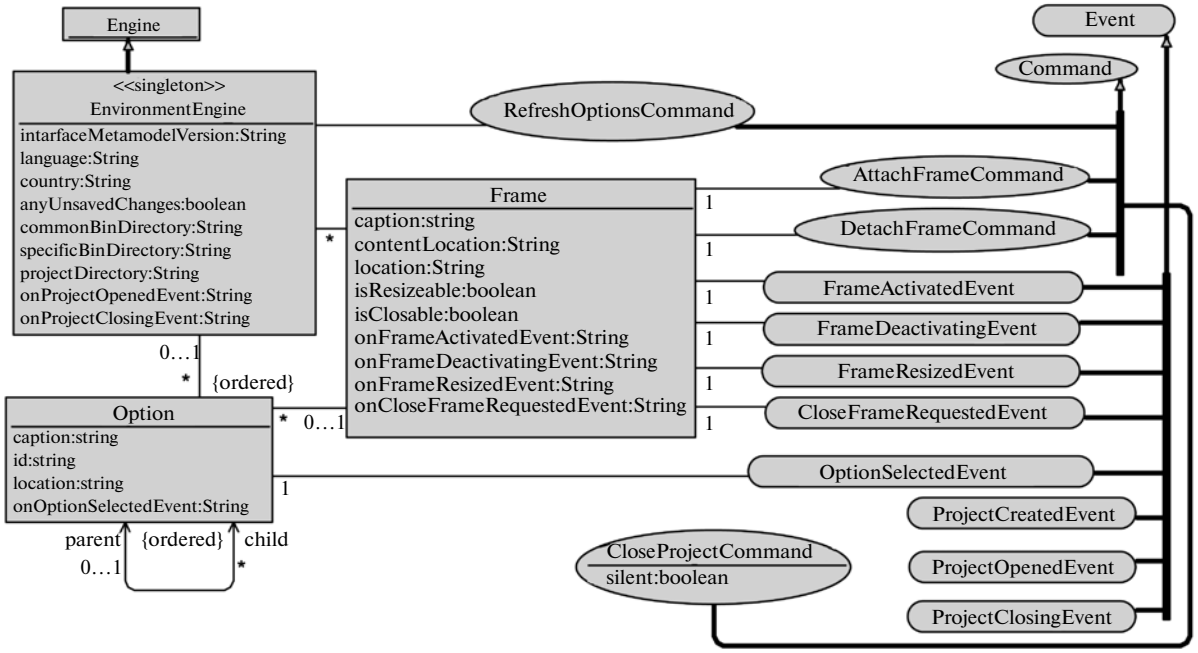

Fig. 6. Environment Engine Metamodel.

ProjectOpenedEvent, or ProjectClosingEvent. The project can be closed also programmatically by submitting the CloseProjectCommand command to the environment engine.

Transformations and engines can add new options via creating objects of the Option class (the location attribute contains the meaning that encodes the location of an option, e.g., in the menu or in the toolbar) and calling the *RefreshOptionsCommand* command. When the user chooses an option, an *OptionSelectedEvent* event is emitted.

If it is required for an engine to show a dialog window, then it creates an object of the Frame class, specifies the location of the window content and the place where the new window should be located inside the main window (see the contentLocation and location attributes; their values are encoded in a specific way). Then, the engine calls the AttachFrameCommand command. When the window becomes active or inactive or when the user changes the dimensions of the window or tries to close it, then corresponding events emerge (the first 4 subclasses of the Event class in Fig. 6). Before closing the window, the engine should detach it from the main window by calling the DetachFrameCommand command.

4. AN ANALOGY BETWEEN TDA AND THE HUMAN BRAIN ARCHITECTURE

Figure 7 shows schematically TDA and the human brain architecture (see, e.g., [28]). As is known, the sense organs transform senses to electric impulses and pass them to certain parts of the brain. This is very similar to how the engines in TDA create models that correpsond to the interface metamodels including the creation of event objects. Almost all the signals (except for smell) pass through the thalamus located in the center of the brain. This is similar to how events and commands pass through TDA Kernel, which calls the required transformations and engines. Transformations which have access to all the interface metamodels can be compared to the prefrontal areas, which unite information from different parts of the brain and are responsible for decision making, resulting in the signal passing to the motor cortex, which is responsible for movement. This can be compared to giving commands in TDA. The hippocampus, which is responsible for accessing the memory, can be compared to RAAPI.

[6] The concept of a project is similar to the concept of a document in a text editor. As a rule, the TDA project is the repository content; however, the project can also contain other external data (e.g., drawings or generated documents).

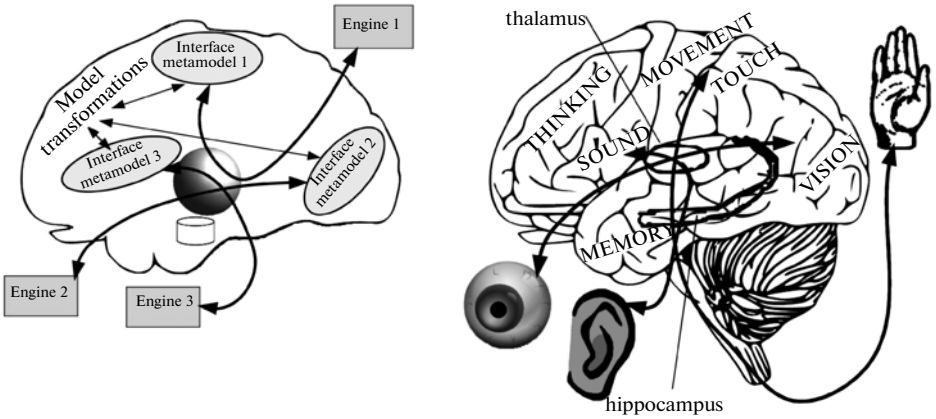

Fig. 7. TDA and architecture of the human brain.

5. CONCLUSIONS

As we have shown in section 2.7, TDA has proved to be advantageous when developing different tools for domain-specific languages based on graph diagrams. We believe that TDA is applicable to the creation of even more complicated systems. This view is supported by the argument that TDA has the association with the architecture of the particularly complex system—the human brain.

ACKOWELEDGMENTS

This work was partially supported by the European Social Fund within the project Support for Doctoral Studies at the University of Latvia, as well as by Latvian National Research Program No. 2 "Development of Innovative Multifunctional Materials, Signal Processing, and Information Technologies for Competitive Science Intensive Products" (project no. 5 "New Information Technologies Based on Ontologies and Model Transformations").

REFERENCES

1. Kleppe, A.G., Warmer, J., and Bast, W., *MDA Explained: The Model Driven Architecture: Practice and Promise,* Boston: Addison-Wesley, 2003.
2. Miller, J. and Mukerji, J., *MDA Guide Version 1.0.1, Document Number: OMG/2003-06-01,* Object Management Group, 2003.
3. Meta Object Facility (MOF) Core Specification Version 2.4.1, Document Number: formal/2011-08-07, Object Management Group, 2011. http://www.omg.org/spec/MOF/2.4.1/PDF
4. Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E., *EMF: Eclipse Modeling Framework,* 2nd Ed., Addison-Wesley, 2008.
5. *Model-Driven Domain Analysis and Software Development: Architectures and Functions*, Osis, J. and Asnina, E., Eds., IGI Global, 2011.
6. *Request for Proposal: MOF 2.0 Query. Views. Transformations, RFP, Document Number: ad/2002-04-10,* Object Management Group, 2002.
7. Kalnins, A., Barzdins, J., and Celms, E., Model Transformation Language MOLA, in *Model Driven Architecture*, Berlin: Springer-Verlag, 2005.
8. Sostaks, A. and Kalnins, A., The Implementation of MOLA to L3 Compiler, *Sci. Papers Univ. Latvia,* 2008, vol. 733, pp. 140–178.
9. Fowler, M., Model Driven Architecture. http://martinfowler.com/bliki/ModelDrivenArchitecture.html
10. Haywood, D., MDA: Nice Idea, Shame about the... http://www.theserverside.com/news/1365166/MDA-Nice-idea-shame-about-the
11. Thomas, D., MDA: Revenge of the Modelers or UML Utopia? *IEEE Software,* 2004, vol. 21, pp. 15–17.
12. Bézivin, J., Why Did MDE Miss the Boat? *Proc. Ist Int. Workshop on Combined Object-Oriented Modeling and Programming (COOMP),* 2011.
13. Jouault, F. and Kurtev, I., Transforming Models with ATL, *Proc. 2005 Int. Conf. on Satellite Events at the Models,* 2006.

14. Kolovos, D., Rose, L., and Paige, R., The Epsilon Book. http://www.eclipse.org/epsilon/doc/book/
15. Agrawal, A., Karsai, G., and Shi, F., Graph Transformations on Domain-Specific Models, 2003. http://w3.isis.vanderbilt.edu/publications/archive/agrawal_a_11_0_2003_graph_tran.pdf
16. Horn, T. and Ebert, J., The GReTL Transformation Language, *Proc. 4th Int. Conf. on Theory and Practice of Model Transformations,* 2011, pp. 183–197.
17. Kalnina, E., Model Transformation Development Using MOLA Mappings and Template MOLA, *Doctoral Thesis,* Univ. Latvia, 2011.
18. OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/
19. OWL 2 Web Ontology Language Document Overview. http://www.w3.org/TR/owl2-overview/
20. Kelly, S. and Tolvanen, J.-P., *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley, 2008.
21. Barzdins, J., Cerans, K., Kozlovics, Rencis, E., and Zarins, A., A Graph Diagram Engine for the Transformation-Driven Architecture, *Proc. of MDDAUI 2009 Workshop of Int. Conf. on Intelligent User Interfaces,* 2009, pp. 29–32.
22. Kozlovics, S., A Dialog Engine Metamodel for the Transformation-Driven Architecture, *Sci. Papers Univ. Latvia,* 2010, vol. 756, pp. 151–170.
23. Sprogis, A., Liepins, R., Barzdins, et al., GRAF: a Graphical Tool Building Framework, *Proc. Tools and Consultancy Track of ECMFA,* 2010.
24. Barzdins, J., Cerans, K., Grismanis, M., Kalnins, A., Kozlovics, S., Lace, L., Liepins, R., Rencis, E., Sprogis, A., and Zerims, A., Domain Specific Languages for Business Process Management: a Case Study, *Proc. DSM'09 Workshop of OOPSLA*, 2009, pp. 34–40.
25. Opmanis, M. and Cerans, K., Multilevel Data Repository for Ontological and Meta-Modeling. Databases and Information Systems VI, *Proc. 9th Int. Baltic Conf., DB and IS,* 2011.
26. Kozlovics, S., The Orchestra of Multiple Model Repositories, *Proc. SOFSEM,* 2013.
27. Kozlovics, S., Rencis, E., Rikacovs, S., and Cerans, K., A Kernel-Level UNDO/REDO Mechanism for the Transformation-Driven Architecture. Databases and Information Systems VI, *Proc. 9th Int. Baltic Conf., DB and IS 2010,* 2011.
28. Carter, R., Aldridge, S., Page, M., and Parker, S., *The Human Brain Book,* Dorling Kindersley, 2009.