Scientific Journal of Riga Technical University
Computer Science. Applied Computer Systems
2010
Volume 43

Implementing a Topological Functioning Model Tool

Armands Šlihte, *Riga Technical University*

***Abstract* - This paper recognizes the computation independent nature of a Topological Functioning Model (TFM) and suggests it to be used as the Computation Independent Model (CIM) within Model Driven Architecture (MDA). To step towards the completeness of MDA and enable the automation of system analysis the Topological Functioning Model for Model Driven Architecture (TFM4MDA) method is considered. A project of implementing TFM4MDA as a TFM Tool is suggested to enable artificial intelligence in system analysis and software development. By applying language processing methods in combination with textual use case analysis and using Topological Functioning Model (TFM) as CIM a workable solution can be developed. The main components of the tool are a TFM Fetcher for system's informal description analysis, TFM Editor and TFM Transformer for TFM to UML transformation. Solution's compatibility to MDA standards is also considered, thus providing a formal method to automatically acquire a CIM from description of a business system in form of textual use cases, and that could be applied to implement a system analysis tool for this task. This paper discusses the implementation challenges and considers implementing the tool by using the leverage of the Eclipse platform and its MDA conformable frameworks.**

Keywords: **graphical modeling framework, language processing, meta-object facility, model driven architecture, topological functioning model**

I. Introduction

Software development is a complex process. Every software development project is unique. However in most cases the abstractions or models of the information systems to be developed may be at least similar if not the same. Software developers are busy with coding similar structures and procedures; the development process becomes somewhat inefficient. Moreover software development is expensive and there are many risks that stakeholders have to take in account. The industry of software development has been approaching and dealing with these issues in different ways.

Model Driven Architecture (MDA) proposes software development to abstract from the code as the uppermost of the functionality of the information system to the model of the information system [1]. That means that first an information system's model is developed and then it is transformed into a ready-to-use information system. Changes and additions also are made using the model. The purpose of MDA is to enable software development using the models of an application and generating the source code from these models.

MDA is a software development framework which defines 3 layers of abstraction for system analysis: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). MDA is based on 4 level architecture and the supporting standards: Meta-Object Facility (MOF), Unified Modeling Language (UML), and XML Metadata Interchange (XMI) [2].

Topological Functioning Model (TFM) offers a formal way to define a system by describing both the system's functional and topological features [3]. TFM is represented in the form of a topological space (X, Θ), where X is finite set of functional features of the system under consideration, and Θ is the topology that satisfies axioms of topological structures and is represented in the form of a directed graph [4]. TFM represents the system in its business environment and shows how the system is functioning, without details about how the system is constructed. TFM4MDA method suggested in [4] and developed in Riga Technical University allows system's TFM to be composed by having knowledge about the complex system that operates in the real world. This paper suggests using TFM as CIM by composing it using TFM4MDA; acquiring a mathematically formal and thus transformable CIM.

The long-term goal of this work is to improve TFM4MDA method and to develop a TFM Tool which would fully implement this method. MDA tools mainly focus on requirements gathering, domain modeling, and code generation [5], not offering a way for defining a formal CIM. This tool starts a new direction of MDA tools by offering construction of a formal CIM and applying elements of artificial intelligence for system analysis and software development. The development of such a tool is a complex and a large scale project, which requires dealing with several issues. This paper talks about the issues related to the task of implementing a TFM Tool.

This paper is organized as follows. Section II analyzes related work, discussing the TFM4MDA method and other approaches dealing with the transformation of an informal description of a system to a formal model. Section III shows the schema of the TFM Tool and identifies its main components. Section IV considers the applied platform and technologies to the development of TFM Tool. Section V examines the challenges of implementing a text analyzer, which would fetch a TFM from an informal description. Section VI describes the development and scope of TFM Tool's component TFM Editor. Section VII discusses further research and the evolution of TFM Tool. Conclusions summarize the work done and explain the significance of further research.

II. Related Work

This work continues research on computation independent modeling and specifically on TFM4MDA started in [3], [4], [6] and [7]. As stated in [4] an informal description of the system in textual form can be produced as a result of system analysis. TFM4MDA is a formal method for transforming this system's informal description into a TFM of the system.

TFM4MDA consists of the following steps: 1) retrieving the system's objects and functional features by analyzing the informal description of a system; 2) constructing a TFM's topological space using the retrieved system's objects and functional features; 3) constructing a TFM's topological graph using its topological space; 4) verifying the functional requirements by mapping them to the corresponding functional features; 5) transforming TFM to UML (probably an UML profile).

To demonstrate how the TFM4MDA method could work in context with TFM Tool an example library system described in [4] is considered. This example will be used throughout the paper. For using TFM4MDA as described in [4] first we need an informal description of a system. Let us consider this fragment: "The librarian checks out the requested book from the book fund to a reader, if the book copy is available in the book fund." This fragment is from [4]. Through text analysis we need to identify system's objects and compose functional features. We can identify the following system's objects: a librarian, a book copy (a synonym is a book), a book fund, a reader. Every functional feature consists of an object action, a result of this action, an object involved in this action, a set of preconditions of this action, an entity responsible for this action, and subordination. Using the given fragment of an informal description we can compose the following functional feature: 1) the action is checking out; 2) the result of this action is a book copy is checked out for a reader; 3) the object involved in this action is a book copy; 4) a precondition of this action is that a book copy has to be available; 5) the entity responsible for this action is a librarian; 6) subordination is inner. TFM Tool will have to be able to retrieve these system's objects and functional features automatically using text analysis.

Next step is to construct a topological space of TFM, meaning that we have to identify the cause-effect relations between the composed functional features. The text analyzer will set the initial cause-effect relations by considering the order in which the functional features appear in the informal description, the results of functional features, the objects involved, and the entity responsible. TFM Tool will then automatically construct a TFM's topological graph.

The tool will then offer the user to correct these initial system's objects, functional features or cause-effect relations, and add new objects or functional features. In addition the tool will enable the user to manually point to the main functional cycle, define functional requirements, and check their conformity to the functional features.

The last step is to transform the TFM to a conformable UML profile. TFM Tool will be able to automatically transform the acquired system's TFM to an UML profile without losing any valuable information.

The approach described in [4] still defines some structure of the informal description, thus making it semi-formal. This paper will introduce more formalism into TFM4MDA's conception of an informal system's description. There have been other attempts to transform an informal description of a system to a formal model. Approach proposed in [9] suggests generating implementation from textual use cases. This approach uses statistical parser on use cases and by analyzing the parse trees compose so called Procases for further use in implementation generation. Procases can be thought of as a formal model of requirements. This paper also suggests textual use cases to be used for defining requirements and as input for text analysis from which a TFM could be composed. Approach described in [9] uses their generated implementation for verifying software requirements and also to use the implementation as a platform to proceed with the development of the software.

III. TFM TOOL'S SCHEMA

TFM Tool is planned to be an implementation of TFM4MDA. The purpose of the tool is to enable users to partly automatically construct a TFM of a system from the system's description in form of textual use cases written in English, verify the functional requirements, and transform the TFM to an UML profile. Use cases describing the requirements of the system have to be developed by the system analyst.

As Fig. 1 shows the TFM Tool's main components are: TFM Fetcher – a tool which fetches the functional features and constructs a TFM's topological space from use cases of a system by applying language processing methods, TFM Editor – a graphical editor for constructing a TFM's topological graph, and TFM Transformer – a XMI transformation tool for transforming TFM to an UML profile. TFM's topological space and topological graph is bound through XMI. Topological graph expands topological space's XMI simply by adding a graphical dimension.

Fig. 1 shows a schema of the TFM Tool. Arrows represent the automatic transformations between artifacts. Grey boxes are the artifacts managed by the tool. White boxes are the TFM tool's components.

The construction of TFM is a step-by-step process. The TFM Tool should allow the user to return to previous steps to make changes and the artifacts of each step should be interconnected. For example, if changes are made in TFM's topological space its topological graph is automatically affected.

Moreover the TFM Fetcher should be able to learn from the actions performed by user, the changes which user makes in the results of the automatic operations of the tool. For example, if by analyzing a system's informal description the tool hasn't recognized correctly some specific functional feature and the user of the tool points to it in the next step, the tool should recognize this functional feature automatically if this informal description is analyzed again. This can be

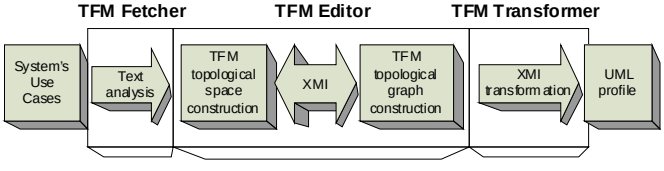

Fig. 1. Schema of the TFM Tool

achieved by applying machine learning methods to the TFM Tool.

The last stage of the TFM Tool's life cycle is TFM transformation to UML. Here is a need for a specific UML profile for TFM so that no useful data from system's TFM is lost. The generation of a fully functioning information system or a significant part of the code of the system task will be past to a conformable tool. More specific explanation about the transformation between TFM and UML is given in [4].

IV. Applied Platform and Technologies

TFM Tool will be developed as an Eclipse plug-in. Eclipse is a software development platform comprising an integrated development environment (IDE) and a plug-in system to extend it [9]. Eclipse equips software developers with a Plug-in Development Environment (PDE) for developing plug-ins. One of the advantages of using Eclipse as the platform for this tool is the large number of contributed plug-ins, including frameworks for MDA and MDA tools.

The first to mention of such frameworks is the Eclipse Modeling Framework (EMF). EMF is a Java framework and code generation facility for building tools and other applications based on a structured model [10]. EMF provides a facility to transform a model into efficient, correct, and easily customizable Java code. EMF defines its models using XMI. EMF can be thought of as a highly efficient Java implementation of a core subset of the MOF, which EMF calls Ecore. There are a number of EMF-based tools and EMF provides full integrity. EMF.Edit is an Eclipse framework that includes generic reusable classes for building editors for EMF models [11]. EMF.Edit can be used to develop additional functionality for the tool generated by EMF generator.

For graphical editor development Eclipse offers a Graphical Editing Framework (GEF). It provides a framework for developing and running an Eclipse graphical editor plug-in. The Graphical Modeling Framework (GMF) improves GEF by integrating it with EMF. Combining these technologies lets a GEF-based graphical editor be developed using model defined with EMF. GMF is divided in two main components: the runtime and the tooling used to generate editors capable of leveraging the runtime [12]. GMF tooling is used to create models, which define the graphical modeling editor, and generate the implementation of the editor. GMF runtime runs the generated graphical modeling editor plug-in.

It is important that this TFM Tool is developed in Eclipse platform because of the possibility to cooperate more effectively with other MDA tools. This is essential for MDA because more likely the complete MDA life cycle will be provided by a set of tools and not only just one. Future community enhancements for EMF or GMF (or any other Eclipse frameworks used in future contributions) will be easily integrated with the TFM Construction Tool. Eclipse is also widely used as an IDE for developing Java applications and information systems. While the MDA life cycle is not complete and the information system's code still might need some customization after generation, the software developer can continue his work using the same environment (Eclipse).

TFM Fetcher will have to use some language parser for working out the grammatical structure of sentences. This is necessary so that TFM Fetcher could analyze the sentences of use cases and retrieve functional features. The Stanford Statistical Parser offers such a parser licensed under the GNU GPL v2, which allows its use for research purposes or other free software projects [13]. The Stanford Parser has made Java libraries available for use.

V. TFM Fetcher

This section talks about the first component of the TFM Tool – TFM Fetcher. The purpose of TFM Fetcher is to fetch a TFM from a system's informal description.

Where does an informal description of the system come from? There are a lot of different methodologies to support software development. All of them require some sort of requirements gathering process, which usually provides software requirements expressed in textual and diagram form. Some of these methodologies are more formal others less formal, but in most cases textual and diagram requirements of the system can be considered as an informal description. Constructing a formal model from text analysis is not a simple task. In a realistic case the description can probably be quite long, incomplete, redundant and inconsistent. To make this task a little easier the description of the system has to have some degree of formality. For this reason use cases are considered.

TFM Tool has to support a number of iterations back and forth between description and TFM Fetcher. The user of the tool should be able to see the mapping between text and TFM and correct any incompleteness, redundancy or inconsistency.

A. System's Informal Description

Use cases are not normalized or standardized by any consortium, unlike UML use case diagram by Object Management Group. Moreover, there are many different use case templates and the structure of a use case can be adjusted depending on the situation and the development team [14]. Usually use case structure can consist of the following or similar sections: use case identifier, description, actors, assumptions, steps, variations and non-functional requirements.

In context of TFM Tool textual use cases are considered the informal description of the system. The following structure of use case is considered: use case title, actors, pre-conditions, main scenario, extensions, and sub-variations. Use case title shortly describes the use case; actors are a list of actors involved in the use case; pre-conditions define the conditions that must be in place before this use case starts; main scenario lists the specific steps (written in natural language) that take place to complete the use case; extensions and sub-variations list deviations from the main success scenario – branch actions, with the difference that extensions are performed in addition to extended action, but sub-variations are performed instead of the extended action. This use case structure is very similar to that proposed in [9]. You can see an example of proposed use case in Fig. 2.

```
Use case: Client is requesting a book
Actors: Client, Librarian
Preconditions: Client has gotten a reader status
Main scenario:
       1. Client fills a book request form
       2. Client gives the filled request form to the
       librarian
       3. Librarian checks out the book from book fund
       4. Librarian gives the book to client
Extensions:
       -
Sub-variations:
       4a. If the book is not availiable in the book fund,
       librarian denies the book request form
```

Fig. 2. Use case "Client is requesting a book"

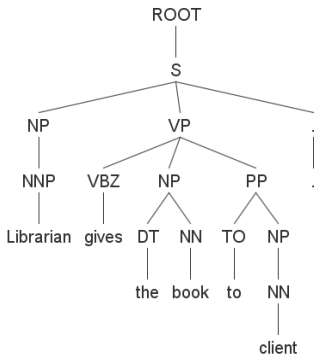

Fig. 3. Parse tree of a use case sentence

B. Retrieving Functional Features

Functional features are represented by a tuple consisting of an object action, a result of this action, an object involved in this action, a set of preconditions of this action, an entity responsible for this action, and subordination. One of the tasks of the TFM Fetcher is to retrieve these functional features from use cases.

Use cases are formed by sentences written in natural language. Every sentence, except title and actors, in a use case can be considered as a representation of a functional feature. Use case sentence can sometimes represent more than one functional feature. This can happen when sentence consists of more than one result of the action or objects involved in the action. Such an issue can be dealt by analyzing sentence's coordinating conjunctions. For example if 1st and 2nd steps of use case from Fig. Y. are combined in one sentence "Client fills the book request form and gives the filled request form to the librarian". In this sentence the second reference to a request form could be replaced by a pronoun "it". This should be taken into account. Moreover, the sentences of a use case should be written as simple and unambiguous as possible, but in realistic case this is not always possible.

Use case's actors will be considered as objects involved in the action and entities responsible for the action. The title can partly be considered as a functional requirement.

TFM Fetcher has to be able to form the corresponding functional features by analyzing the use case sentences. For this purpose natural language processing methods have to be applied.

Concrete syntax tree or parse tree will be used for the analysis of use case sentences. Parse tree is a tree that represents the syntactic structure of a sentence according to some formal grammar [15]. Parse trees are usually output of parsers, which can use different methods for finding the right parse tree for the specific sentence. The most efficient parsers are statistical parsers which associate grammar rules with probability. For example, use case sentence "Librarian gives the book to client" will be parsed using The Stanford Parser [13]; results are shown in Fig. 3.

In Fig. 3 the abbreviations are Part-Of-Speech tags according to [16]: S – sentence, NP – noun phrase, VP – verb phrase, NNP – proper noun, singular, VBZ – verb, 3rd person singular present, DT – determinant (article), NN – noun singular or mass, PP –prepositional phrase, TO – "to".

First a functional feature description has to be identified. In this case it is the verb phrase of the sentence in present continuous tense – "Giving the book to client". It consists of the object action (give), the result of the action (book) and object involved in the action (client). The responsible entity for the action can be determined by comparing the actors list of the use case and the noun phrase. In this case the noun phrase is "Librarian" and there is "Librarian" in the actors list as well, so the entity responsible for the action probably is "Librarian". Preconditions can be determined by analyzing preconditions and sub-variations of the use case. If the current functional feature is represented as the first step of use case main scenario, then one additional precondition will match the precondition of the use case itself. If current step has a sub-variation, then the functional feature represented by the next step will have a precondition that is the opposite of the sub-variation condition. For example, sub-variation "If book is not available in the book fund, librarian denies the book request" will result in a precondition "Book is available" for functional feature "Giving the book to reader". Use case extensions define their own precondition; obviously the condition in the extension's sentence is the precondition of the functional feature represented. Functional feature's subordination can be determined only by the user of the TFM Tool.

By analyzing use case sentences it should be possible to derive functional features. It is important that TFM Fetcher considers functional features the same if they are represented by the same tuple. This means that no duplicate functional features are created and two or more use cases can include the representation of the same functional features.

C. Composing TFM

Once there is a set of functional features it is necessary for TFM Fetcher to retrieve the topology of TFM or cause-effect relations between functional features. The structure of use cases will help with this task.

Firstly, every use case's main scenario is an ordered sequence of functional features. Additionally, by analyzing the extensions and sub-variations it is possible to detect branching in a TFM. Extension adds an effect to the functional feature represented by the step referenced by the extension. On the

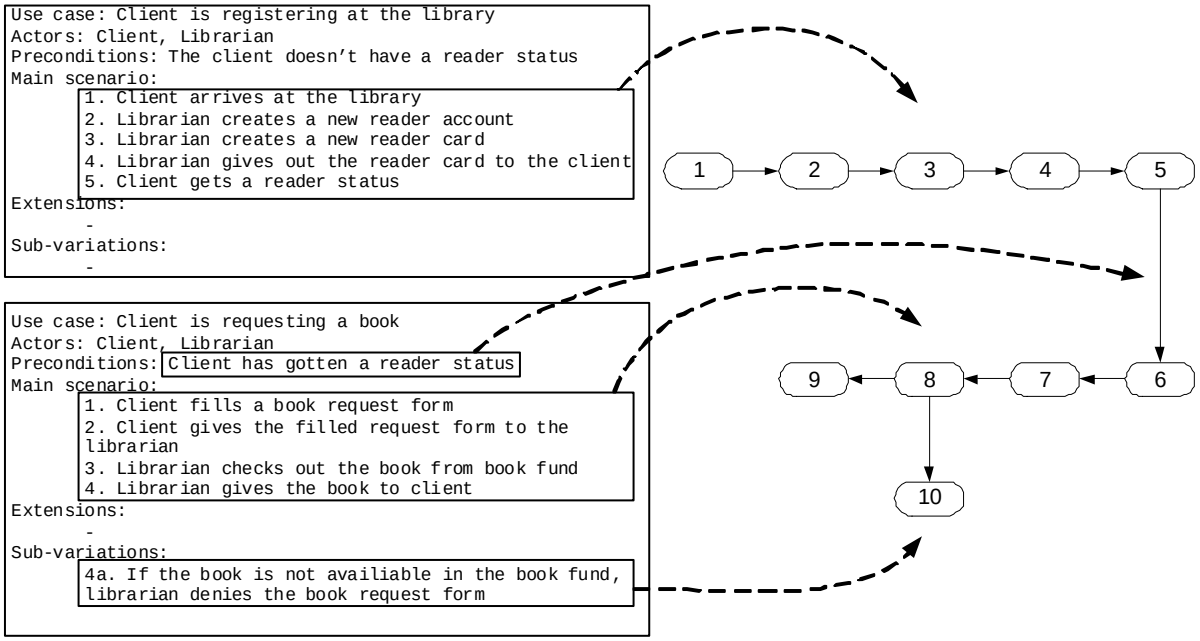

Fig. 4. Fetching TFM's topology

other hand, sub-variation adds an effect to the functional feature represented by the previous step referenced by the sub-variation. Therefore, the setting of cause-effect relations between functional features represented within the same use case is very straightforward. As you can see in Fig. 4 the 2 sequences of functional features come from main scenarios of use cases. The 8th functional feature has an additional effect "Denying the book" because of the sub-variation.

Another thing is setting the cause-effect relations between functional features fetched from different use cases. Precondition section of use cases are used to define this relation. If a use case's step represented by a functional feature results in a condition defined in another use case's preconditions, then this other use case's first step functional feature is the effect of the first functional feature. For example, in Fig. 4 the 2nd use case has a precondition "Client has gotten a reader status". By analyzing this precondition it can be detected that the verb phrase of the sentence is the present perfect tense of "getting a reader status" which is a description of 5th functional feature. But this will not always work because precondition can be declared in different tense. For example, "Client is at the library". The corresponding description of functional feature would be "being at the library". Nevertheless, by having this conclusion it is possible to set the 6th functional feature as the effect of the 5th functional feature. This is one of the hardest tasks of the TFM Fetcher.

VI. TFM Editor

This section describes the scope and development of TFM Editor. A prototype is developed using the Eclipse platform, Eclipse Modeling Framework and Graphical Modeling Framework.

TFM4MDA's first step is retrieving the system's objects and functional features by analyzing the informal description of a system. But first this analyzer would need a certain data structure according to which to store its results – system's objects and functional features. The first thing this tool needs is some sort of framework within which to operate and evolve. The prototype of TFM Editor must provide such a framework. To do this, prototype will implement the core functionality of the TFM Tool, namely the construction of TFM's topological space and topological graph (as shown in Fig. 1). The prototype will provide users with functionality to manually construct a TFM's topological space with the system's objects and functional features, construct the TFM's topological graph in addition to this topological space, and verify functional requirements. In more detail, the prototype will enable users to define functional features and indicate the cause-effect relations between them, point to the system's main cycle, define functional requirements and indicate the compliance with functional features. For developing such a prototype the domain metamodel and a TFM model editor must be developed, thus creating the main data structures and a framework for the prototype to use and also for the future contributions to use.

The first step to developing a graphical modeling editor is defining the metamodel of the domain. In the case of TFM Editor the domain is TFM itself. To develop a TFM graphical modeling editor with GMF a metamodel of TFM has to be created with compliance to XMI.

Fig. 5 shows the created TFM metamodel. Within the scope of the TFM Editor's prototype TFM will consist of system main cycle, functional features, and functional requirements. Each functional feature comprises the following attributes: object action, result of this action, the object involved in this action, set of preconditions of this action, entity responsible for this action, and subordination. Functional features can be in cause-effect relations with each other. Cycle element is for specifying the main cycle of the system. Functional

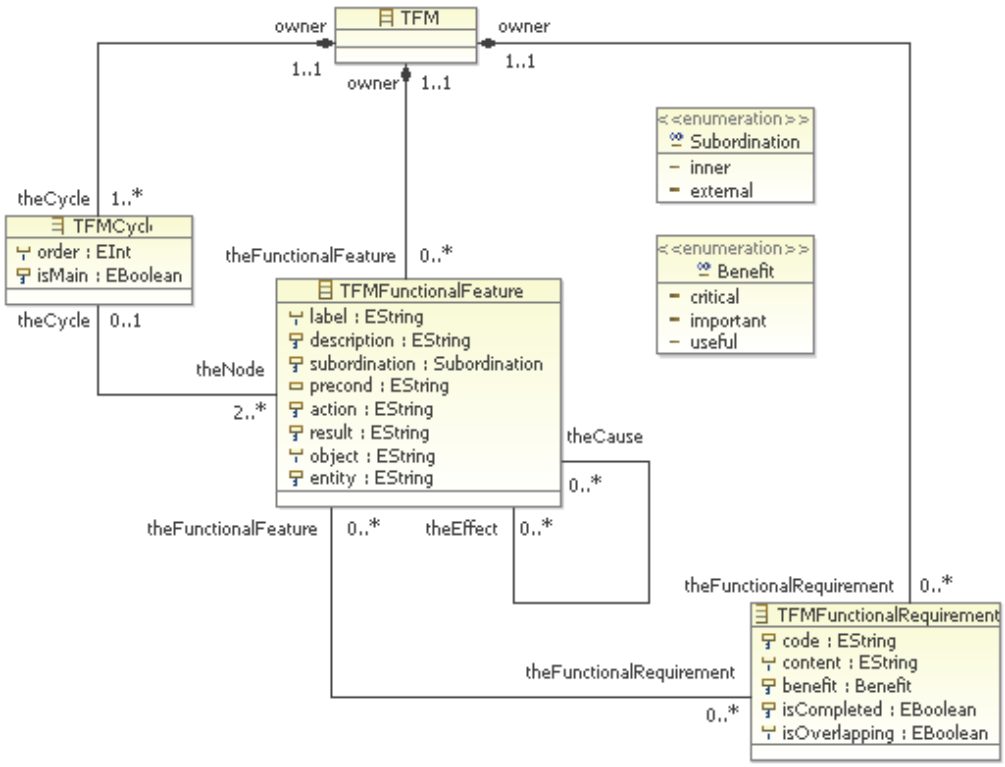

Fig. 5. MOF-compatible metamodel (XMI) of TFM

requirement attributes are: code, content, which describes the requirement, business benefit, whether it is complete, and whether it is overlapping. Subordination can be inner or external. Benefit can be critical, important or useful.

TFM metamodel is created using EMF's Ecore format which basically is plain XMI. At this point it is already possible to generate a TFM editor, but the goal is to acquire an Eclipse based graphical modeling editor of TFM. To do this GMF is applied. GMF requires a set of models to be able to generate the TFM graphical modeling editor.

The creation of these models is a step-by-step process and they have to be created in the following order: 1) TFM metamodel; 2) EMF model editor generator model; 3) GMF graphical definition model, which defines the elements of the diagram, labels and figure gallery; 4) GMF tooling definition model, which defines the palette and menus; 5) GMF mapping definition model, which maps the graphical and tooling definitions; 6) GMF graphical modeling editor generator model. GMF provides the necessary tools to create these models. The GMF graphical modeling editor generator model is the resulting model from which the TFM graphical modeling editor can be generated.

Thus the TFM Editor's prototype is generated as an Eclipse plug-in. This acquired Eclipse plug-in depends on EMF, GMF runtime, GEF, and Eclipse platform.

The acquired TFM Editor main components are the palette, canvas and properties pane. In addition to standard GMF features the tool enables user to define functional features and indicate the cause-effect relations between them, point to the system's main cycle, define functional requirements and indicate the compliance with functional features.

TFM Editor is ready for constructing TFMs. When creating a new TFM, the tool creates 2 files: the TFM model file (extension .tfm) and the TFM diagram file (extension .tfm_diagram). Both of them use XMI for storing metadata. These files are interconnected. Any changes made in the model affect the diagram and vice versa. The fact that basically the model is stored in XMI means that it can be easily exported or used in another MDA tool for further transformations.

By generating this tool using GMF the source code of the tool is available for enhancements and future contributions. It is possible to add some custom code using special Java annotations and regenerate the tool without losing the custom code. Thus a fully functional TFM Editor's prototype is created and ready for use. The important thing is that now there is a framework and a head start to continue work on implementing the TFM4MDA.

VII. Further Research

Further research is related to the evolution of the TFM Tool bringing its functionality closer to TFM4MDA. First thing in queue is implementing the TFM Fetcher for automatically retrieving system's objects and functional features from its informal description and composing a TFM. Thus elements of artificial intelligence will be used in purpose for software development.

TFM Fetcher should be able to retrieve object actions, results of these actions, the objects involved in these actions, preconditions of these actions, entities responsible for the actions and form functional features with cause-effect relations between them. This paper discusses the issues related

to retrieving functional features and cause-effect relations between them, thus constructing a TFM. An important task is to develop a formal algorithm for fetching TFM using language processing.

These initially retrieved functional features will be available for editing with TFM Editor. The TFM metamodel will probably have to be adjusted for this purpose, but that is not an obstacle because, as mentioned earlier, the tool can be regenerated at any time without losing any custom code. The functional feature elements should be associated with the specific parts of the use cases and changes applied after modifications in the TFM. This would ensure that a TFM can be regenerated using this set of use cases.

TFM Fetcher will probably not be able to be correct every single time, and the user will have to correct its mistakes. Moreover it should be able to learn from its mistakes by using machine learning methods and so become more efficient.

Another thing to do is to develop a TFM Transformer which would transform TFM to UML. As mentioned before it will probably be a special UML profile to keep all the valuable information of the TFM. So firstly there is a need for a specific TFM UML profile. Secondly Eclipse offers UML2 and UML2 Tools which can be applied for dealing with TFM Tool's problem of TFM to UML profile transformation. From this point it should be possible to generate some part of the system's code.

VIII. Conclusions

This paper discusses TFM4MDA and TFM construction, applying language processing for fetching TFM, Eclipse platform and its options of application generation from metamodels in MDA context, development of TFM Editor using Eclipse technologies.

Nowadays software developers often are occupied with similar pattern application coding. MDA proposes to abstract from application source code to the model of the application as the main artifact in software development. Until now in MDA context everyone has his own opinion about what is a CIM. This paper suggests that TFM should be considered as the CIM of a system and it should be constructed using TFM4MDA. Thus a mathematically formal and transformable CIM of a system is acquired.

This paper talks about the challenges of retrieving a formal model from informal description of the system represented by use cases, and applying language processing methods to deal with these issues. A fully functional prototype of the TFM Editor was developed using Eclipse platform and its MDA facilities. This prototype will serve as a framework for evolvement of the TFM Tool and future contributions. This paper clearly shows that the future research of TFM4MDA branches in 3 main tasks the development of the TFM Fetcher, UML profile for TFM, and TFM Transformer for a TFM to UML profile transformation.

With advancements of this TFM Tool research the completeness of MDA will improve. TFM4MDA provides a formal CIM and new horizons by partially automating and improving system analysis, and introducing artificial intelligence to software development.

References

[1] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. Indianapolis: Wiley, 2003.

[2] D. Gasevic, D. Djuric, and V. Devedzic, *Model Driven Architecture and Ontology Development*. Heidelberg: Springer, 2006.

[3] J. Osis, "Topological model of system functioning," (in Russian), in *Automatics and Computer Science*. Riga: Zinatne, 1969, pp. 44-50.

[4] J. Osis, E. Asnina, and A. Grave, "Computation independent representation of the problem domain in MDA," *J. Software Eng*, vol. 2, no. 1, pp. 19-46. [Online]. Available: http://www.e-informatyka.pl/wiki/e-Informatica_-_Volume_2.

[5] M. Kontio, "Architectural manifesto: choosing MDA tools," IBM. [Online]. Available: http://www.ibm.com/developerworks/library/wi-arch18.html. [Accessed: visited October 2009].

[6] E. Asnina, "The formal approach to problem domain modeling within model driven architecture," in *9th International Conference on Information Systems Implementation and Modeling.* Czech Republic, Ostrava: Prerov, 2006, pp. 97-104.

[7] J. Osis and E. Asnina, "Enterprise modeling for information system development within MDA,""in *41th Annual Hawaii International Conference on System Sciences.* USA: HICSS, 2008, pp. 490.

[8] J. Francu and P. Hnetynka, "Automated generation of implementation from textual system requirements," in *Proceedings of the 3rd IFIP TC 2 CEE-SET*. Brno, Czech Republic: Wroclawskiej, 2008, pp. 15-28.

[9] "EMF developer guide: the Eclipse modeling framework (EMF) overview," The Eclipse Foundation. [Online]. Available: http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html. [Accessed: October 2009].

[10] "EMF developer guide: the EMF.Edit framework overview," The Eclipse Foundation. [Online]. Available: http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.Edit.html. [Accessed: October 2009].

[11] F. Plante, "Introducing the GMF runtime." [Online]. Available: http://www.eclipse.org/articles/Article-Introducing-GMF/article.html. [Accessed: October 2009].

[12] "The Stanford Parser: a statistical parser," The Stanford Natural Language Processing Group. [Online]. Available: http://nlp.stanford.edu/software/lex-parser.shtml. [Accessed: October 2009].

[13] R. Malan and D. Bredemeyer, "Functional requirements and use cases," Bredemeyer Consulting. [Online]. Available: http://www.bredemeyer.com/pdf_files/functreq.pdf. [Accessed: October 2009].

[14] D. Jurafsky and J. H. Martin, "Speech and language processing: an introduction to natural language processing," in *Computational Linguistics, and Speech Recognition*. Pearson Education, 2000, p. 908.

[15] B. Santorini, "Part-of-speech tagging guidelines for the Penn Treebank Project," 3rd revision, 2nd printing, Penn Treebank Project. [Online]. Available: http://www.ldc.upenn.edu/Catalog/desc/addenda/LDC1999T42/TAGGUID1.PDF. [Accessed: October 2009].

Armands Šlihte was born April 1, 1986 in Riga, Latvia. In year 2008 he graduated Riga Technical University, Riga, Latvia with bachelor's degree of engineering science in computer control and computer science.

Author is currently working as a Scientific Assistant in Faculty of Computer Science and Information Technology, Institute of Applied Computer Systems, Riga Technical University, Riga, Latvia.

Armands Šlihte. Topoloģiskā Funkcionēšanas Modeļa rīka izstrāde

Šajā rakstā tiek analizētas topoloģiskā funkcionēšanas modeļa (TFM) no skaitļošanas neatkarīgās iezīmes un tiek piedāvāts no skaitļošanas neatkarīgs modelis (CIM) modeļu vadāmās arhitektūras (MDA) kontekstā. Lai pilnveidotu MDA un padarītu iespējamu sistēmu analīzes automatizāciju, topoloģiskās funkcionēšanas modelim tiek piedāvāts izmantot modeļu vadāmās arhitektūras (TFM4MDA) metodi. Rakstā tiek piedāvāts izstrādāt TFM TFM4MDA atbalsta rīku, tā padarot iespējamu mākslīgā intelekta izmantošanu sistēmu analīzē un programmatūras izstrādē. Pielietojot dabīgās valodas apstrādes metodes kombinācijā ar biznesa lietošanas gadījumiem un TFM kā no skaitļošanas neatkarīgo modeli ir iespējam izstrādāt praktiski pielietojamu risinājumu. Rīka galvenās komponentes ir sistēmas neformālā apraksta analizators, TFM konstruēšanas rīks, TFM transformācija uz UML. Tiek apskatīta arī risinājuma atbilstība

MDA standartiem, tādējādi piedāvājot formālu metodi, ar kuru iespējams iegūt formālu CIM no biznesa sistēmas apraksta lietošanas gadījumu formā. Tādā veidā šo metodi ir iespējams arī realizēt atbilstošā sistēmas analīzes rīkā. Rīka izstrādei tiek izsvērta Eclipse platformas piedāvātas iespēju MDA ietvaru izmantošana, kā arī apskatītas šāda rīka izstrādes problēmas.

Армандс Шлихте. Разработка инструментария модели топологического функционирования

В статье проведен анализ от вычислений независимых свойств модели топологическо функционирования (TFM) и предложена от вычислений независимая модель (CIM) в контексте управляемой моделью архитектуры (MDA). Для усовершенствования MDA и получения возможности автоматизировать системный анализ, для модели топологического функционирования предлагается использовать метод управляемой моделью архитектуры (TFM4MDA). В статье предлагается разработать инструмент поддержки TFM TFM4MDA, получая возможность использовать искусственный интеллект в системном анализе и разработке программ. Главными компонентами инструментария являются анализатор неформального описания системы, инструмент конструирования TFM, трансформатор TFM в UML. Рассматривается также и соответствие этого решения стандартам MDA, таким образом предлагая формальную модель, с помощью которой возможно получить формальную CIM исходя из описания бизнес системы в форме частных приложений. Данный метод также можно реализовать и в соответствующих средствах системного анализа. Для разработки инструментария проверяется использование возможностей платформы Eclipse для MDA, а также рассмотрены проблемы разработки такого инструментария.