RECONCILING SOFTWARE REQUIREMENTS AND ARCHITECTURES WITHIN MDA

PROGRAMMATŪRAS PRASĪBU UN ARHITEKTŪRAS SASKAŅOŠANA MODEĻVADĀMAJĀ ARHITEKTŪRĀ

U.Donins, J.Osis

Software requirements, architecture, model driven architecture, topological modelling

1. Introduction

In the software development world little guidance and few methods are available for reconciling software requirements and architecture which satisfies those requirements. Reconciliation of software requirements and architectures is a process in which is developed software architecture according to requirements. The reconciling process, if performed accordingly, ensures that developed architecture satisfies defined software requirements.
The main goal of this paper is to define an approach by which it is possible to reconcile software requirements and architectures within model driven architecture (MDA). MDA considers system from three viewpoints: computation independent viewpoint, platform independent viewpoint and platform specific viewpoint. Each viewpoint has its own model by which the viewpoint is modelled. These models are computation independent model (CIM), platform independent model (PIM) and platform specific model (PSM). Topological functioning model (TFM) allows modelling functioning of a system and defining and preserving details of concepts of the system under construction and it can be used as a representation of MDA's computation independent model. Because TFM is a formal model it is possible to use it to reconcile software requirements and architectures and to make formal transformation from computation independent model into platform independent model. The use of TFM provides possibility for traceability between software artefacts, e.g. between requirements and architecture elements. By using case study we have proven that it is possible to reconcile requirements and architectures by using TFM. The software architecture in this case is modelled by using new type of class diagrams – topological class diagrams. At the end of the case study we have shown how we can introduce more formalism into Unified Modelling Language (UML) diagrams by transforming topology from TFM to class diagrams.
This paper is organized as follows. Section 2 describes the key principles and suggested solutions of reconciling software requirements and architectures. Section 3 discusses the use of TFM for reconciling process. By using TFM in the modelling process it is possible to introduce topology even in the class diagrams. As a result we have constructed a new type of class diagrams – topological class diagrams. Description of the problem domain modelling is illustrated with an example which clearly shows all obtained software development artefacts. Section 4 gives conclusions of our work.

2. Methods for reconciling software requirements and architectures

It is possible to reconcile software requirements and architectures in several ways. One of the methods for reconciling process is to use different models for describing the software requirements, for example, using goal models for software requirements modelling and then obtaining architecture directly from these goal models. Another way for reconciling is to use intermediate models, where pre-architecture elements are defined according to requirements, and develop architecture from intermediate model. Fig.1. depicts development of software architecture directly from requirements and requirements' models by using intermediate models.

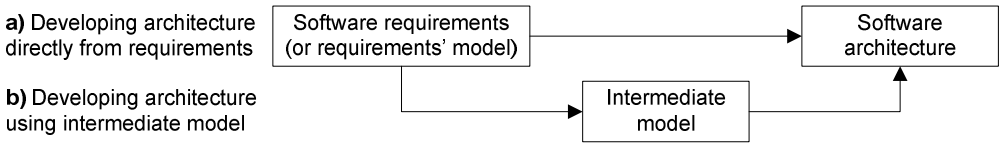

Fig.1. Relations between requirements and architecture

Typical stages of software development are requirements analysis, designing, and development [1]. The development of the software architecture begins already in the requirements analysis stage and ends in the designing stage. If we are looking from the model driven architecture (MDA) [2] viewpoint, then software architecture is platform independent model (PIM), but requirements model is computation independent model (CIM). Several methods useful in reconciling software requirements and architectures, that is methods used for transformation from CIM to PIM, are described, in brief, in the sections below.

2.1. Development of the software architecture by using requirements' models

One of the approaches for modelling software requirements that we can mention is goal oriented approach KAOS [3][4][5]. By using KAOS models it is possible to model both functional and non-functional requirements. In the context of KAOS approach is defined method for obtaining software architecture basing on the developed KAOS models [6]. By using this method as the architecture development beginning is taken software specification which can be obtained from KAOS models. For each agent defined in KAOS model is developed components, which are related with system goals achieving in which are under development. Interfaces for these components are developed according to the sets of variables which are managed and controlled by agents. Then data flow connectors is developed for every combination from two components where one component controls variable which is monitored by agent. In this way the initial architecture is developed. After the initial architecture development the obtained model is restructured according to one of the chosen architecture styles. Architecture styles in detail are described in [7].
In the described software architecture development approach the architecture model is obtained directly from the goals model by using relations between goals to make data connectors. This means that initial architecture can be automatically generated from the goals model [8]. In the [8] is stated that the development of software architecture by using goal oriented approach is one of the active research field in goal oriented requirements engineering.

2.2. Development of the software architecture by using intermediate models

One of the approaches which suggest using intermediate models in the reconciling process is CBSP (Component-Bus-System-Property) approach [9]. Name of the developed intermediate model is the same as for the approach – CBSP. This approach is developed based on the twin peaks model [10] in which software requirements and architecture is developed iteratively and in the same time.
Requirements in the CBSP model are represented using six types (dimensions) of the CBSP elements which involve the basic architectural constructs: *C* (model elements that describe or involve an individual *C*omponent in architecture); *B* (model elements that describe or imply a *B*us (connector)); *S* (model elements that describe *S*ystem-wide features or features pertinent to a large subset of the system's components and connectors); *CP* (model elements that describe or imply data or processing *C*omponent *P*roperties. The properties in CBSP are the "ilities" in a software system, for example, reliability, scalability, and adaptability); *BP* (model elements that describe or imply *B*us *P*roperties); and *SP* (model elements that describe or imply *S*ystem (or subsystem) *P*roperties).
By using CBSP model software requirements are connected with architecture elements (for example, components) where the CBSP model serves as connector between requirements and architecture. Relations between requirements, CBSP model and architecture are shown in Fig.2.

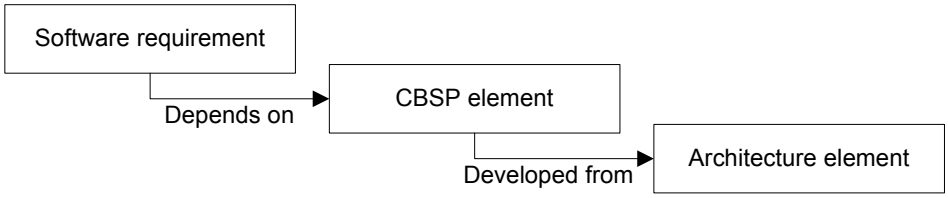

Fig. 2. Relations between requirements, CBPS model and architecture

The authors of CBSP approach have defined five steps for developing CBSP model: *1* – selection of requirements for next iteration; *2* – architectural classification of requirements; *3* – identification and resolution of classification mismatches; *4* – architectural refinement of requirements; and *5* – trade-off choices of architectural elements and styles with CBSP.
The developed CBSP intermediate model is like proto-architecture which suggests the appropriate architectural style and helps in further architecture development.

2.3. Development of the software architecture by using topological functioning model

According to the modified MDA life cycle [11] it is possible to reconcile software requirements and architecture by using topological functioning model (TFM).
TFM has strong mathematical basis and is represented in a form of a topological space (X, Θ), where X is a 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 a form of a directed graph. TFM has topological (connectedness, closure, neighbourhood, and continuous mapping) and functional (cause-effect relations, cycle structure, and inputs and outputs) characteristics. It is acknowledged that every business and technical system is a subsystem of the environment within the system functions.
There are defined three steps for the TFM construction for problem domain modelling in a business system context (see Section 3.1).
Cause-effect relations between functional features form causal chains that are functioning cycles. All the cycles and subcycles should be carefully analyzed in order to completely identify existing functionality of the system. The main cycle (or cycles) of system functioning (i.e. functionality that is vitally necessary for system's life – if the main cycle of system is destroyed or interrupted then system can no longer function or exist) must be found and analyzed before starting further analysis. In case of studying a complex system, a TFM can be separated into a series of subsystems according to identified cycles.
In the [12] and [13] are demonstrated possibilities for construction of use case diagrams and non-oriented conceptual class diagrams by using TFM. In this way it is tried to formalize CIM within the MDA. This formalization can help to generate the PIM with the use of CIM transformation. In the [14] was introduced the idea of using TFM within the context of MDA for CIM modelling with the goal to make MDA software development more formal within the very beginning of the software modelling and development.
TFM also can be considered as intermediate model in the reconciling process (because initially from requirements is constructed TFM and only after construction of TFM the initial software architecture is developed). Comparing TFM with CBSP model we can see that TFM is a formal model and its use is more convenient. With the help of TFM is represented functioning of the system under consideration but the CBSP approach endeavours to represent software architecture by using CBSP model elements.
The use of TFM for the reconciling of software requirements and architecture in detail are described in next section.

3. Reconciling software requirements and architectures within MDA

To reconcile software requirements and architecture it is needed to construct TFM (the construction of TFM is described in detail in [11]) and to use modified TFMfMDA (Topological functioning modelling for MDA) approach. Unmodified TFMfMDA approach is described in [12]. Schematic representation of reconciling process is given in Fig. 3, where rectangles denote miscellaneous software development artefacts (for example, software requirements, TFM), rectangles with rounded edges denote processes, arrows denote data flows between processes and labels on the arrows denote data which flows from one process to another.

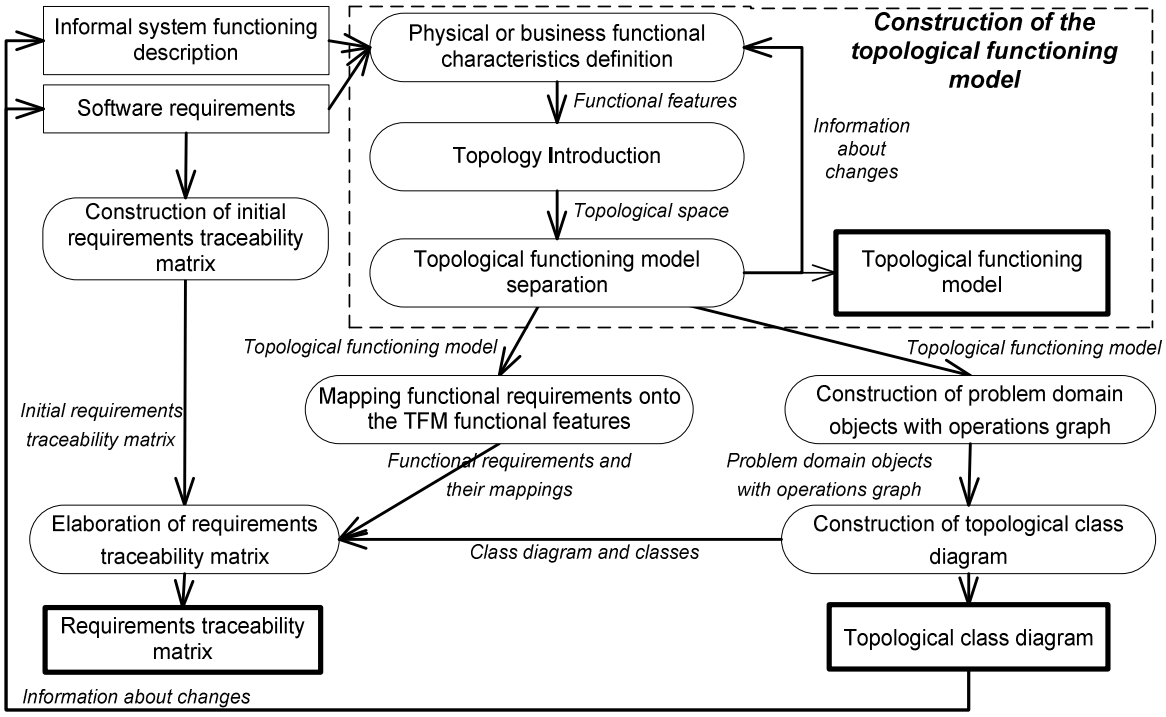

Fig. 3. Process of reconciling software requirements and architectures within MDA

TFMfMDA approach represents software architecture in the form of non-oriented conceptual class diagram, which contains conceptual classes and non-oriented associations between them. TFMfMDA suggests introducing orientation of associations between classes within developed class diagram in the further analysis of the business system. If directions of associations are introduced in this way then it is needed to return to the beginning of the analysis and to review already developed artefacts (for example, informal system description, requirements, TFM). The reviewing process consumes extra time which can be economized by using idea of topological UML (topUML) diagrams published in [15]. By combining the idea of topUML and TFMfMDA approach it is possible to introduce more formalism in the class diagrams and to construct topological class diagrams, where topology between classes is introduced from the developed TFM.
In folowing subsections is in detail described how to reconcile software requirements and architecture within model driven architecture by using TFM according to the processes given in Fig. 3.

3.1. Construction of the topological functioning model

Construction of TFM, which is represented in the form of topological space (X, Θ), consists of following three steps [11] (see Fig. 3 section “Construction of the TFM”):

1. definition of physical or business functional characteristics (inner and external objects, functional features) by means of noun and verb analysis in the informal problem description;
2. introduction of topology, i.e. establishing cause effect relations between functional features (these relations are represented as arcs of a directed graph that are oriented from a cause vertex to an effect vertex); and
3. separation of the topological functioning model from the topological space.

After the construction of the TFM it is possible to map requirements onto the TFM and functional features. The mapping can be made in several ways. One of the ways is to use requirements traceability matrix [16] and the other way is to use mappings defined in the TFMfMDA approach [12]. These mappings serve also as traceability between defined requirements and developed artefacts.

In this subsection below is described the use of TFM in the reconciling process in the context of library system case study. As mentioned in [17] library system as example is convenient and better understandable because every one knows something about library and its functioning.

Below is given informal description from the project, in which a library application is developed.

„All library visitors are registered. Registration is done by the receptionist. Any visitor, who is registered in the library readers' register and who has filled out and filed the reader's card is considered as a reader. If the visitor is not a registered reader yet, the receptionist performs the reader registration. If the visitor does not have the reader's card yet or has lost his/her reader's card, the receptionist makes it anew. The reader's card bears the information about the reader and every reader's card has its own unique identification number. Registered readers with reader cards have the right to use the library catalogue in order to find the book they need. Only one catalogue is available in the library. Each catalogue entry contains the title of the book, author's name, the publisher, the publishing date, the ISBN code (if any). To borrow a book from the library, the reader has to complete the request form and enter the code, the title and the author of the desired book. Having completed the request form, the reader submits it to the librarian, who counts the number of books already borrowed by the reader. If the number of borrowed books does not exceed the maximum allowed number, the librarian checks, if the reader's chosen book is available from the library repository. If the chosen book is available from the library repository, the librarian hands out the book to the reader. The library has only one book repository.

When the reader returns the book, the librarian checks its condition. If the book is damaged, the librarian calculates the fine and issues the fine ticket to the reader, who later pays the fine. If the book is extremely damaged and cannot be used anymore, the librarian withdraws it and delivers to utilizer, and, if this is the only copy of the book, also removes it form the library catalogue. If the book is in good condition, the librarian puts it back into book repository, registers as returned and makes it available for borrowing. The reader can only perform one activity at a time – either borrow new books, or return the already borrowed ones.

When the library buys a book, the receptionist assigns it a new identification number. If the book is not entered in the library catalogue yet, the receptionist registers it in the catalogue. After the book registration, receptionist places it in the book repository and makes it available for borrowing."

For the library example are defined the following requirements: **FR1**: The system has to provide the registration of new readers and new reader card preparation (also for the existing readers, if the previous card has been lost); **FR2**: Only registered readers and card holders shall be provided with the access to book catalogue; **FR3**: The system has to provide the procedure for book hand out to readers, including the request form preparation; **FR4**: The system has to provide the procedure for book return from the readers, including the checking of book condition; **FR5**: The system has to provide the fine calculation for damaged books and preparation of fine tickets as well as have to provide the possibility to check that the given ticket has been paid up; **FR6**: The system has to provide, that extremely damaged books after they are returned to the library, are not placed in the book repository; **FR7**: The system has to provide the new book placement into book repository, and if necessary: **FR8**: Prior to a book transfer to utilization, the book utilization request form must be filled out., into book catalogue.

Step 1: Definition of physical or business functional characteristics (functional features) is done according to specified informal system description and defined functional requirements. Within [18] is suggested that each functional feature is a tuple $<A, R, O, PrCond, E>$, where A is an object action, R is a result of this action, O is an object (objects) that receives the result or that is used in this action

(for example, a role, a time period, a catalogue, etc.), *PrCond* is a set PrCond = {c1, …, ci}, where ci is a precondition or an atomic business rule (it is an optional parameter), and E is an entity responsible for performing actions. Each precondition and atomic business rule must be either defined as a functional feature or assigned to an already defined functional feature. We have added two more elements to the tuple described above and these elements are: *Cl* and *Op*, where Cl is a class that represents current functional feature and Op represents operation which is responsible for performing action defined by this functional feature.

Functional features of the TFM and system cycles are defined by the experts. The defined functional features for the library example are as follows (where Reg denotes Registrar, R – Reader, L – Librarian, Ex – External, and In – Inner): <**1**, Arrival of the visitor, Ø, Visitor, Ex, Ø, Ø>, <**2**, Verification of the visitor's personal data in the readers register, Ø, Reg, In, Ø, Ø>, <**3**, Registration of the visitor in the readers register, If the visitor is not registered, Reg, In, Ø, Ø>, <**4**, Preparation of the reader card, If the reader has no reader card (or) If the reader has lost the reader card, Reg, In, Ø, Ø>, <**5**, Reader's card issue to the reader, Ø, Reg, In, Ø, Ø>, <**6**, Verification of the reader's status, If the reader is registered (and) If the reader has reader card, R, In, Ø, Ø>, <**7**, Searching for books in the catalogue, If the reader has reader card, R, In, Ø, Ø>, <**8**, Filling of the book request form, If the reader has found necessary book, R, In, Ø, Ø>, <**9**, Submitting of the book request form, Ø, R, In, Ø, Ø>, <**10**, Verification of the reader's took books count, Ø, L, In, Ø, Ø>, <**11**, Verification of the book availability in the books repository, If the took books count is lower than the allowed maximum, L, In, Ø, Ø>, <**12**, Taking out book copy from books repository, If the book copy is available in the books repository, L, In, Ø, Ø>, <**13**, Checking out book copy, Ø, L, In, Ø, Ø>, <**14**, Taking out book copy, Ø, R, In, Ø, Ø>, <**15**, Returning book copy, Ø, R, Ex, Ø, Ø>, <**16**, Checking condition of the book copy, Ø, L, In, Ø, Ø>, <**17**, Calculation of the fine, If the book copy is damaged, L, In, Ø, Ø>, <**18**, Giving out fine bill, Ø, L, In, Ø, Ø>, <**19**, Paying of the fine, Ø, R, Ex, Ø, Ø>, <**20**, Returning book copy to books repository, Ø, L, In, Ø, Ø>, <**21**, Retiring book copy, If the book copy is hardly damaged, L, In, Ø, Ø>, <**22**, Taking out book from catalogue, If retired last book copy, L, In, Ø, Ø>, <**23**, Purchasing new book, Ø, Library, Ex, Ø, Ø>, <**24**, Inputting book data in the catalogue, If this is first book copy of this book in library, Reg, In, Ø, Ø>, <**25**, Assigning identification number to the book copy, Ø, Reg, In, Ø, Ø>, <**26**, Utilization of the book, If the book is hardly damaged, Utilizer, Ex, Ø, Ø>, <**27**, Updating books repository, Ø, L, In, Ø, Ø>, <**28**, Filling of the book utilization form, Ø, L, Ex, Ø, Ø>, <**29**, Closing the fine, If the reader has paid fine, L, In, Ø, Ø>.

All above defined tuples of the functional features does not contain yet values for elements f and g (classes and operations). These values will be defined later during the definition of software architecture.

Step 2: After finishing definition of functional features the next step is introduction of topology between defined functional features. As a result of this action is developed topological space of the system functioning. Topological space is represented in a form of directed graph where each vertex is a functional feature but arcs between them are topology (cause and effect relationships between functional features). In the Fig. 4.a is represented topological space of library functioning.

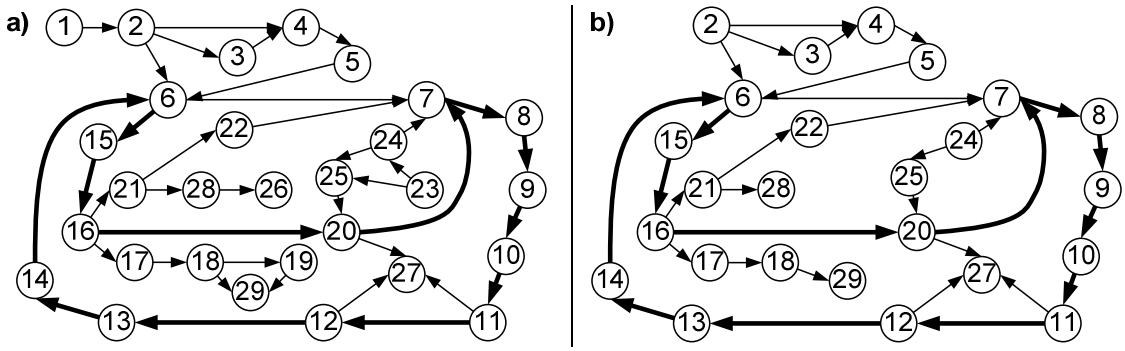

Fig. 4. Both topological space of the library functioning (a) and TFM of the library functioning (b)

The topological space in Fig. 4.a represents the main functional cycle defined by the expert, which includes the following functional features "6-15-16-20-7-8-9-10-11-12-13-14-6" and is denoted by bold lines. These functional features describe checking out and taking back a book. These are assumed to be the main, because they have a major impact on the operation of the business system. A cycle that includes the functional features "6-7-8-9-10-11-12-13-14-6" illustrates an example of the first-order subcycle.

Step 3: After the construction of topological space it is necessary to perform separation of the TFM. Separation is performed by applying the closure operation over a set of system's inner functional features [11]. Topological space is a system represented by Equation (1), where N is a set of inner system functional features and M is a set of functional features of other systems that interact with the system or of the system itself, which affect the external ones.

$$Z = N \cup M \tag{1}$$

$$X = [N] = \bigcup_{\eta=1}^{n} X_{\eta} \tag{2}$$

A TFM ($X \in \Theta$) is separated from the topological space of a problem domain by the closure operation over the set N as it is shown by Equation (2), where $X\eta$ is an adherence point of the set N and capacity of X is the number n of adherence points of N. An adherence point of the set N is a point, whose each neighbourhood includes at least one point from the set N. The neighbourhood of a vertex x in a directed graph is the set of all vertices adjacent to x and the vertex x itself. It is assumed here that all vertices adjacent to x lie at the distance d=1 from x on ends of output arcs from x. Moreover, a TFM can be separated into a series of subsystems by the closures of chosen subsets of N.

The example illustrates how is performed the closuring operation (2) over the set N in order to get all of the system's functionality – the set X:

- The set of the system's inner functional features N = {2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22, 24, 25, 27, 29}.
- The set of external functional features and system functional features that affect the external environment M = {1, 6, 18, 19, 21, 23, 26, 28}.
- The neighbourhood of each element of the set N is as follows: X_2 = {2, 3, 4, 6}, X_3 = {3, 4}, X_4 = {4, 5}, X_5 = {5, 6}, X_7 = {7, 8}, X_8 = {8, 9}, X_9 = {9, 10}, X_{10} = {10, 11}, X_{11} = {11, 12, 27}, X_{12} = {12, 13, 27}, X_{13} = {13, 14}, X_{14} = {14, 6}, X_{15} = {15, 16}, X_{16} = {16, 17, 20, 21}, X_{17} = {17, 18}, X_{20} = {20, 7, 27}, X_{21} = {21, 28}, X_{22} = {22, 7}, X_{24} = {24, 7, 25}, X_{25} = {25, 20}, X_{27} = {27}, and X_{29} = {29}.
- The obtained set X = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 24, 25, 27, 29}.

Obtained TFM after performing closuring operation is shown in Fig. 4.b. The example represents the main functional cycle defined by the expert, which includes the following functional features "6-15-16-20-7-8-9-10-11-12-13-14-6" and is denoted by bold lines. These functional features describe checking out and taking back a book. These are assumed to be the main, because they have a major impact on the operation of the business system. A cycle that includes the functional features "6-7-8-9-10-11-12-13-14-6" illustrates an example of the first-order sub-cycle.

After performing the closuring operation the construction of TFM is completed.

3.2. Software requirements mapping onto the topological functioning model

By using TFM within the reconciling process it is possible to use traceability approach defined within TFMfMDA, which uses mappings of requirements onto functional features. Mappings of requirements are described with arrow predicates. An arrow predicate is a construct borrowed from the universal categorical logic. Universal categorical (arrow diagram) logic for computer science is explored in detail in [19]. By using arrow predicates it is possible to define relations between requirements and

functional features. According to [13] there are five types of mappings and corresponding arrow predicates defined for mapping requirements onto TFM: **One to One** (see Fig. 5.a); **Many to One** (see Fig. 5.b and Fig. 5.c); **One to Many** (see Fig. 5.d and Fig. 5.e); **One to Zero**; **Zero to One**.
The requirements map onto the features as follows: FR1={2, 3, 4, 5}; FR2={6, 7}; FR3={8, 9, 10, 11, 12, 13, 14, 27}; FR4={15, 16, 20, 27}; FR5={17, 18, 29}; FR6={21, 22}; FR7={24, 25, 27}; and FR8={28}. The corresponding representation of requirement mappings is shown in Fig. 5 (f).

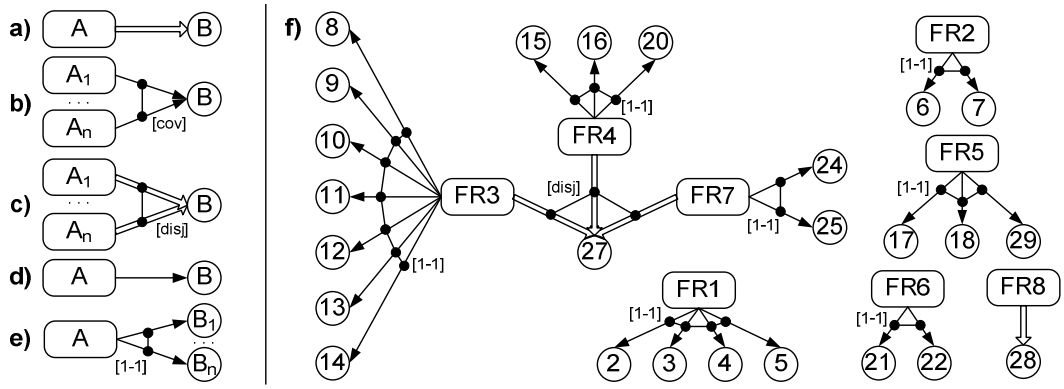

Fig. 5. Functional requirement mappings onto the TFM functional features

Another way to provide traceability between developed software artefacts is to use requirements traceability matrix (RTM) [15] which are initially developed in the beginning of the software development and supplemented during all the software life cycle. Rows in the RTM represents requirements but columns – developed artefacts.

3.3. Development of the software architecture

In the [13] is offered development of conceptual class diagrams as the final step of the TFM usage. In these conceptual class diagrams relevant information – directions of the relations between the classes – is lost. It is recommended to define those relation directions in further software development, for example, to develop a more detailed software design. But at this point a step back should be taken to review the TFM and its transformation on the conceptual class diagram (conceptual classes with no directed associations between them). To avoid such regression and to save the obtained topology between the classes it is possible to develop a topological class diagram where the established TFM topology between classes is retained. Development of topological class diagrams is possible by using the idea published in [15] about topological UML (TopUML) diagrams (including also topological UML class diagrams).
Topological relations between classes throughout this article are marked with discontinuous lines where in the line interruptions are placed two points. Example of topological relations is shown in Fig. 6.a.

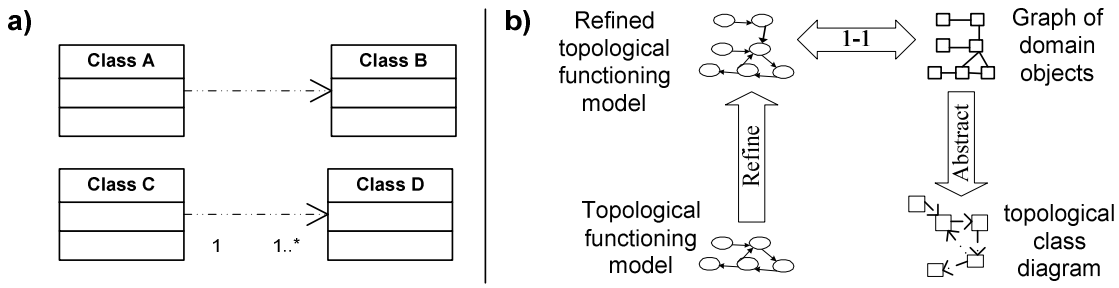

Fig. 6. Topological relations between classes (a) and process of development of the topological class diagrams (b)

In order to obtain a class diagram, first of all a graph of problem domain objects must be developed and afterwards transformed into a class diagram. In order to obtain a problem domain object graph, it is necessary to detail each functional feature of the TFM to a level where it uses only one type of objects. After that this more accurate model must be transformed one-to-one to a problem domain object graph and then the vertices with the same type of objects and methods must be merged, while keeping all relations with other graph vertices. As a result, object graph with direct associations is defined. Schematic representation of class diagram development is given in Fig. 6.b.

By using the ideas published in [13] it is possible to obtain a conceptual class diagram from TFM without orientated relations between classes and the classes without operations. By modifying this approach it is possible to develop not only topological class diagrams, where the direction of associations is retained, but also to obtain the possible class operation definitions. In order to define conceptual operations, it is necessary to detail not only each functional feature to one kind of object, but also by doing this detailed elaboration, to add a operation to the obtained object (using a point notation). Operation is description which shortly describes the defined activity of the functional feature. For example, the functional feature "*Reader's card issue to the reader*" is detailed to the object "*ReaderCard*" with operation "*GiveOutToReader()*" (when point notation is used the obtained result looks like this: "*ReaderCard.GiveOutToReader()*").

Our example skips the step of the topological functioning model refinement, because each functional feature deals only with one type of objects and operations. Fig. 7 shows the transformation of the topological functioning model to the graph of domain objects with conceptual operations.

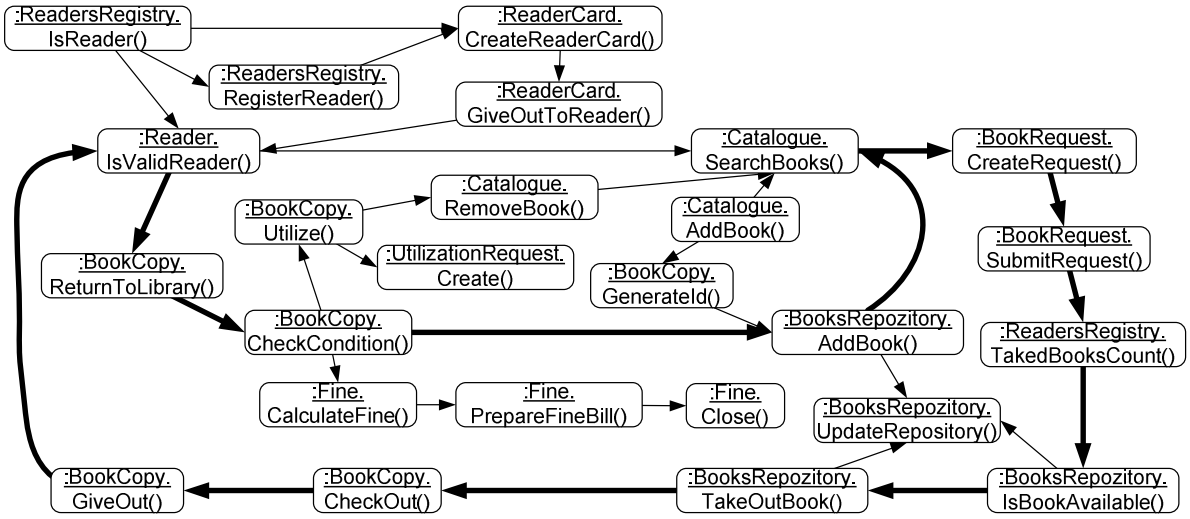

Fig. 7. The graph of domain objects with operations

Elements *Cl* and *Op* of the functional feature tuple for functional features defined in section 3.1 initially were left empty. After construction of object graph it is possible to fill these elements. For example, for the functional feature *13* the value for the *Cl* element is "*BookCopy*", but for the *Op* element – "*CheckOut*". The final definition of the functional feature *13* is as follows: *<13*, *Checking out book copy*, *Ø*, *B*, *In*, *BookCopy*, *CheckOut()>*.

After construction of the object graph it is possible to develop topological class diagram. Fig. 8 presents topological class diagram of the library example after domain object graph is abstracted, i.e., after merging all graph vertices with the same object types. With the bold lines in developed topological class diagram is represented the main functional cycle which was defined within constructed TFM.

The topological relations between classes, which are described with a one-way arrow, cannot be compared with none of the UML relations between the classes given in language specification. The UML language specification [20] gives the following relations between the classes: dependence, association, aggregation, composition, generalization and realization.

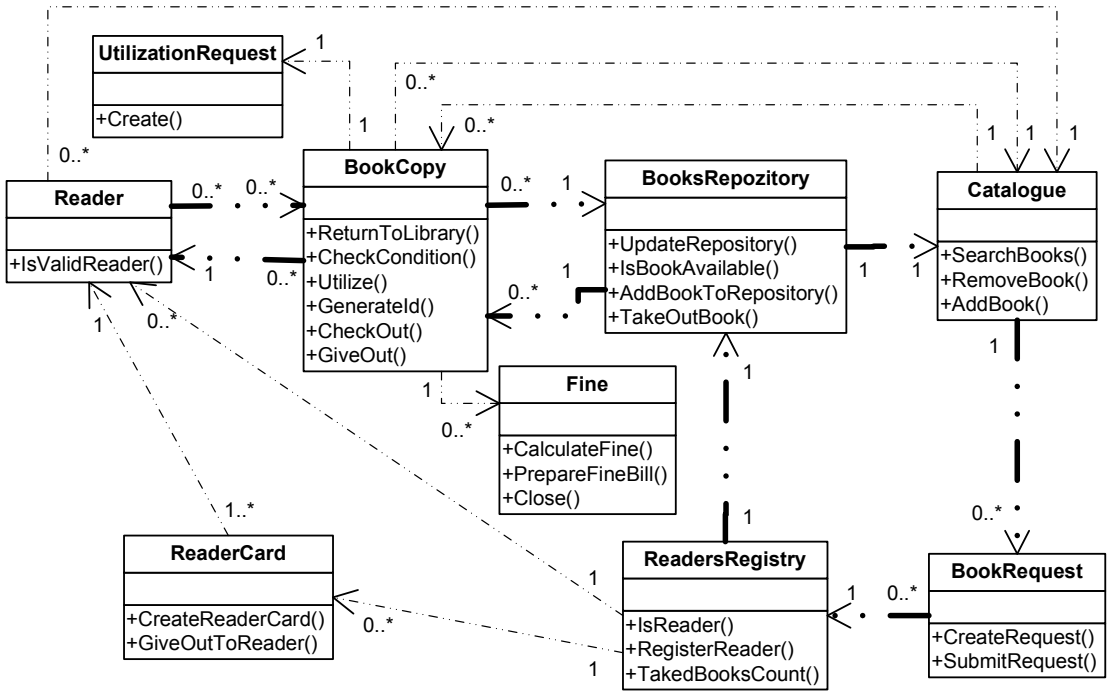

Fig. 8. Topological class diagram

All previously mentioned relations between classes define the way classes interact and use each other, but the adopted topology in UML class diagram allows keeping the substantiation as to the cause of each class, for example, the book can cause the issue of a fine ticket rather than the fine can cause book. The saved topology between classes in class diagram enables more efficient development of the software system class diagram.

By keeping topological relations between the classes it is recommended to use in one-way directed relation, because two mutually opposed relations between two classes can represent various multiplications. If topological class diagram is used to make the non-oriented UML conceptual class diagram, then relations between two classes can be joined into one, and as a multiplicity save the biggest multiplicity of all topological relations between those two classes.

4. Conclusions

The application of the TFM for requirements and architectures reconciling within MDA has the following advantages: 1) by using TFM it is possible formalize and make more precise reconciling process and obtain architecture which satisfies defined software requirements; 2) with the help of TFM is introduced more formalism in the UML class diagrams and in their construction.

The retained topology in class diagrams brings more formalism in these class diagrams. Formalism of class diagrams is improved because between the classes now are precisely defined relations. In traditional software development relations between classes are defined by the modeller's discretion (the approach given in the [13] helps to identify associations between classes but the identification of direction for these relations again are defined by the modeller's discretion).

In our work we have shown that it is possible to maintain in the class diagrams the topology which is developed using TFM. By the means of MDA it is transformation from CIM to PIM. This research provides further transformations of the MDA models. Using TFM it is also possible to provide traceability between software requirements, functional features and developed architecture elements.

The further research directions are intended to supplement the architectural models that will be developed with the system dynamical performance representation models (for example, activity diagrams, which can also implement topology – similarly to topological class diagrams described in

the paper) and to develop methodology for creating informal description of system functioning. To continue working on topological UML diagrams, it is necessary to supplement the description of topological class diagrams, to create the meta-model of the topological class diagram as well as to study the possibilities of topology implementation into other UML diagrams and to assess its influence on the software system development.

References

1. Hay D. Requirements Analysis: From Business Views to Architecture. – USA: Prentice Hall, 2003. – P.496.
2. MDA Guide Version 1.0.1. / OMG, 2003. – www.omg.org/docs/omg/03-06-01.pdf. – 2003.06.01.
3. Dardenne A., van Lamsweerde A., Fickas S. Goal-Directed Requirements Acquisition // Science of Computer Programming. – Volume 20, Issue 1-2 (1993), P.3-50.
4. From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering / edited by – van Lamsweerde A., Letier E. // Berlin: Springer, 2004. – P.325-340.
5. CEDITI. A KAOS Tutorial / CEDITI, 2003. – http://www.objectiver.com/download/documents/KaosTutorial.pdf – 2003.
6. From System Goals to Software Architecture / edited by – van Lamsweerde A. // Berlin: Springer, 2003. – P.25-43.
7. Fielding R. Architectural Styles and the Design of Network-based Software Architectures. – USA: University of California, 2000. – P.180.
8. Lapouchnian A. Goal-Oriented Requirements Engineering: An Overview of the Current Research. – Toronto: Department of Computer Science University Of Toronto, 2005. – P.32.
9. Grünbacher P., Egyed A., Medvidovic N. Reconciling software requirements and architectures with intermediate models // Software and Systems Modeling. – Volume 3, No 3. (2004), P.235-253.
10. Nuseibeh B. Weawing together requirements and architectures // IEEE Computer. – Volume 34, No 3. (2001), P.115-117.
11. Osis J. Formal computation independent model within the MDA life cycle // International Transactions on Systems Science and Applications. – Volume 1, No. 2. (2006), P.159-166.
12. Osis J., Asnina E., Grave. A. Computation Independent Modeling within the MDA // Proceedings of the IEEE International Conference on Software Science, Technology and Engineering. – Herzlia, Israel, 2007. – P.22-34.
13. Osis J., Asnina E. Enterprise Modeling for Information System Development within MDA // Proceedings of the 41st Hawaii International Conference on System Sciences (HICSS 2008). – Washington DC: IEEE Computer Society, 2008. – P.490.
14. Osis. J. Software Development with Topological Model in the Framework of MDA // Proceedings of the 9th CAiSE/IFIP8.1/EUNO International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'2004) in connection with the CAiSE'2004. – Vol. 1. (2004) – P.211.-220.
15. Osis J. Extension on Software Development Process for Mechatronic and Embedded Systems // Proceedings of the 32nd International Conference on Computers and Industrial Engineering. – Limerick, Ireland: University of Limerick, 2003. – P.305-310.
16. Texel P., Williams C. Use cases combined with Booch/OMT/UML: Process and Products. – USA: Prentice Hall PTR, 1997. – P.465.
17. Briand L., Labiche Y. A UML-Based Approach to System Testing // Software and Systems Modeling. –Volume 1, No. 1. (2002), P.10-42.
18. E. Asnina. The Formal Approach to Problem Domain Modeling within Model Driven Architecture // Proceedings of the 9th International Conference on Information Systems Implementation and Modelling. – Prerov, Czech Republic, Ostrava, 2006. – P.97-104.
19. Universal Arrow Foundations for Visual Modeling / edited by – Diskin Z., Kadish B., Piessens F., Johnson M. // Berlin: Springer, 2000. – P.323-334.
20. Unified Modeling Language Infrastructure v2.1.2 / OMG, 2007. – http://www.omg.org/cgi-bin/apps/doc?formal/07-11-03.pdf – 2007.

Uldis Donins, Mg.sc.ing., researcher, Department of Applied Computer Science, Institute of Applied Computer Systems, Riga Technical University, Meza 1/3, Riga, LV 1048, Latvia, phone: +371 26323094, uldis.donins@cs.rtu.lv

Janis Osis, Dr.habil.sc.ing., professor, Department of Applied Computer Science, Institute of Applied Computer Systems, Riga Technical University, Meza 1/3, Riga, LV 1048, Latvia, phone: +371 67089523, janis.osis@cs.rtu.lv

Doniņš U., Osis J. Programmatūras prasību un arhitektūras saskaņošana modeļvadāmajā arhitektūrā
Programmatūras izstrādes pasaulē ir pieejamas nedaudzas vadlīnijas un metodes, kas domātas programmatūras prasību un arhitektūras saskaņošanai, pie tam, tādā veidā, lai izstrādātā arhitektūra atbilstu definētajām prasībām. Praktiski visas programmatūras prasību un arhitektūras saskaņošanas metodes neizmanto matemātisku pamatu saskaņošanas procesa veikšanai. Raksta galvenais mērķis ir definēt pieeju, kā saskaņot programmatūras prasības un arhitektūru modeļvadāmajā arhitektūrā. Modeļvadāmā arhitektūru programmatūras sistēmu apskata no trīs dažādiem skatu punktiem. Katrs no šiem skatu punktiem tiek attēlots ar savu modeli. Prasību un arhitektūras saskaņošanu modeļvadāmajā arhitektūrā ir iespējams veikt izmantojot topoloģisko funkcionēšanas modeli, kā arī izmantojot topoloģisko funkcionēšanas modeli ir iespējams veikt no skaitļošanas neatkarīga modeļa formālu transformāciju uz platformas neatkarīgu modeli. Topoloģiskā funkcionēšanas modeļa izmantošana saskaņošanas procesā nodrošina trasējamību starp dažādiem programmatūras artefaktiem, piemēram, starp programmatūras prasībām un arhitektūras elementiem. Raksta ietvaros ar praktiska piemēra palīdzību ir demonstrēts, ka programmatūras prasības un arhitektūru ir iespējams saskaņot saskaņošanas procesā izmantojot topoloģisko funkcionēšanas modeli. Ar praktiskā piemēra palīdzību ir parādīts, ka ir iespējams formalizēt UML diagrammas, ieviešot UML diagrammās topoloģiju, ir apskatīta topoloģijas ieviešana klašu diagrammā un topoloģiska klašu diagramma.

Donins U., Osis J. Reconciling software requirements and architectures within MDA
In the software development world little guidance and few methods are available for reconciling software requirements and architecture which satisfies those requirements. In fact none of these methods use formal basis for the reconciling process. The main goal of this paper is to define an approach by which it is possible to reconcile software requirements and architectures within model driven architecture. Model driven architecture considers system from three viewpoints. Each viewpoint has its own model by which the viewpoint is modelled. It is possible to use topological functioning model of system to reconcile software requirements and architectures and to make formal transformation from computation independent model into platform independent model. The use of topological functioning model provides possibility for traceability between software artefacts, e.g. between requirements and architecture elements. By using case study we have proven that it is possible to reconcile requirements and architectures by using topological functioning model. The software architecture in this case is modelled by using topological class diagrams. At the end of the case study we have shown how we can introduce more formalism into UML diagrams by transforming topology from topological functioning model to class diagrams.

Донынш У., Осис Я. Согласование требований и архитектуры программного обеспечения в управляемой моделями архитектуре MDA
В мире разработки программного обеспечения существует несколько рекомендаций и методов, которые можно использовать для согласования требований и архитектуры, удовлетворяющей этим требованиям. Практически ни один из этих методов не использует формальную базу для согласования. Главная задача данной статьи – определить подход, способный согласовать программные требования и архитектуру в рамках управляемой моделями архитектуры. Управляемая моделями архитектура рассматривает систему с трёх точек зрения. Каждой точке зрения соответствует своя модель. Существует возможность использования топологической модели функционирования системы для согласования программных требований и архитектуры, а также для реализации формальной трансформации независящей от вычислений модели в модель, назависящую от платформы. Применение топологической модели функционирования в процессе согласования обеспечивает трассируемость между артефактами программного обеспечения, например, между требованиями и элементами архитектуры. На примере, приведённом в статье, мы доказали, что существует возможность согласования требований и архитектуры, используя топологическую модель функционирования. В этом случае архитектуру программного обеспечения можно моделировать, используя топологические диаграммы классов. В конце примера мы показали, как, путём трансформации топологии в топологических диаграммах классов, в диаграммы UML можно внедрить дополнительный формализм.