doi: 10.1515/acss-2015-0012

Applied Computer Systems

2015/17

The Model Transformation for Getting a UML Class Diagram from a Topological Functioning Model

Arturs Solomencevs, *Riga Technical University, Latvia*

Abstract **– The approach called Topological Functioning Modeling for Model Driven Architecture (TFM4MDA) uses a Topological Functioning Model (TFM) as a formal Computation Independent Model (CIM) within the Model Driven Architecture (MDA). The object of this research is the construction of a UML class diagram on the Platform Independent Model (PIM) level in conformity with the TFM. Nowadays this transformation is executed manually. Manual creation of models is time-consuming and there is a risk of making mistakes. These drawbacks increase expenses and reduce efficiency of TFM4MDA approach. That is why automation of transformation is useful. The paper presents an algorithm for the transformation which is written in a pseudocode and can be implemented as a tool.**

Keywords **– Algorithm for automatic model transformation, model driven architecture, topological functioning model, UML class diagram.**

I. Introduction

Model Driven Architecture (MDA) is an approach to system development, which increases the power of models in this study. The purpose of MDA is to separate the views and concerns. MDA has three viewpoints and their corresponding models: a Computation Independent Model (CIM) contains knowledge about the problem domain and the requirements for software system; Platform Independent Model (PIM) focuses on the operation of a system while hiding the details necessary for a particular platform; and Platform Specific Model (PSM) [1]. Model transformation forms a key part of MDA. To get the software source code, we need to go by the path CIM → PIM → PSM → source code.

Topological Functioning Model (TFM) is a formal model, which describes the functioning of system. The TFM has a solid mathematical base. The model-driven software development approach called Topological Functioning Modeling for Model Driven Architecture (TFM4MDA) is based on the TFM [2]. TFM4MDA introduces a more formal analysis and modeling of the problem domain within the MDA [3], [4]. TFM within the MDA is used as a CIM.

Since the TFM is a formal model, its usage has the following benefits:

- Possibility of transformation to the PIM (within the MDA);
- Guarantee that a software product completely satisfies functional requirements;
- Design process and code generation can be at least partially automated;
- The correctness of operation of the entire system is mathematically proven.

The object of this research is transformation from the TFM to a Unified Modeling Language (UML) class diagram [5] on the PIM level. UML class diagram is important in software development, because it displays the structure of the software system and indicates class responsibilities. Nowadays the creation of a class diagram from the TFM requires fully manual execution. Manual execution is time-consuming; also there is a probability that a user (e.g., a system architect) will make a mistake during the execution. Time investment and risk of making mistakes increase the costs of software development. The costs must be minimized. Therefore, the goal of the research is to automate the transformation from the TFM to a UML class diagram. The algorithm of automated transformation is developed. There is a possibility to develop a tool that will execute the transformation algorithm. As a result of transformation, the initial UML class diagram (with attributes, operations and without relationships among classes) on the PIM level is constructed.

The paper is structured as follows. Section II describes related research – other software development approaches (apart from TFM4MDA) that include the creation of CIM. In Section III, the TFM, MDA and TFM4MDA are described in more detail. In Section IV, the creation of class diagram from the TFM is described. In Section V, the transformation algorithm from the TFM to a UML class diagram is introduced. In Section VI, conclusions are presented.

II. Related Research

There are different approaches for domain modeling that include the creation of CIM. Since model transformation is a key part of MDA, we are interested in approaches that give an opportunity to create a class diagram on the PIM level from the CIM.

Business Process Modeling and Notation (BPMN) is an Object Management Group (OMG) standard [6]. BPMN is used for modeling the problem domain within the Business Process Modeling approach. BPMN model is positioned on the CIM level within the MDA [7]. BPMN can be transformed to a UML activity diagram on the CIM level, and the activity diagram can be transformed to a class diagram on the PIM level. However, a conclusion is made that the gap between BPMN and UML is too large so the creation of an activity diagram from BPMN model is limited under some situations [8]. Not all BPMN elements can be transformed without the loss of information or meaning.

ArchiMate is an Open Group Standard, which provides a graphical language for the representation of enterprise architectures [9]. A CIM is created at the ArchiMate business layer. A Meta Object Facility meta-model [10] for the ArchiMate language does not exist today [11]. It means that the formal transformation from an ArchiMate CIM to a UML class diagram on the PIM level does not exist.

A development approach that is supported by a tool named a Use Case Driven Development Assistant (UCDA) allows converting the functional requirements into a class model semi-automatically. The functional requirements are specified and represented by use cases [12], [13]. Thus, the use case model is used as a CIM. Using a use case model as a CIM is disputable, because it is fragmentary. By calling the model “fragmentary” we mean that it consists of separate fragments and it is not holistic. The fragmentary nature of the model has several shortcomings. There is no way to tell whether the model is complete. Furthermore, it can be hard to check whether there are no conflicts (the bigger the model, the harder to check). Therefore, a use case model is not applicable as a CIM for modeling big systems. This drawback is shared by other software development approaches that are driven by use case modeling. Comparing to the TFM, a use case model lacks formalism. The disadvantage of using a use case model is discussed in more detail in Section III.

A methodology and a tool, Linguistic assistant for Domain Analysis (LIDA), provide linguistic assistance in the model development process. The goal of this method is to utilize existing text descriptions of a problem domain, and from them, produce an initial conceptual class diagram with attributes, methods and roles [14]. The conceptual class diagram is a PIM level model. Prior to using the methodology, the analyst should already have prepared a set of use cases or scenarios that represent the operational concept for the proposed system [14]. Thus, the LIDA helps with analyzing texts (e.g., documents, descriptions of problem domain), but the analyst has to identify which classes are relevant based on the prior developed use case model. Hence, use cases take place as a CIM within the LIDA approach. Therefore, the LIDA approach is driven by use case modeling and has the same drawback discussed in the previous paragraph.

Semantics of Business Vocabulary and Business Rules (SBVR) is another OMG specification that defines the vocabulary and rules for documenting the semantics of business vocabularies and business rules for the exchange among organizations and between software tools [15]. An approach to transform the SBVR model to a UML class diagram on the PIM level is introduced [16]. The process has limitations. The authors are not able to find out the input parameters of class methods. For this moment this drawback also appears within the TFM4MDA approach (in transformation to a class diagram). As far as the author of this paper understands, the SBVR model is fragmentary. Hence, it has the same drawbacks as the use case model.

In the Natural Language Based Requirements Analysis (NIBA), the textual requirement specifications are firstly linguistically analyzed and translated into the so-called conceptual predesign schema – Klagenfurt Conceptual Predesign Model (KCPM) [17]. KCPM provides a user (stake-holder) centered form or requirement documentation, which means that the model can be understood and validated by the users [18]. KCPM can be considered a CIM, because it represents the knowledge about the problem domain, it is used for obtaining the requirements for software, and it is understandable by the end-user [18]. KCPM can be mapped to a UML class diagram [18]. A drawback of NIBA approach is that the requirements must be written in the German language so that they could be automatically analyzed and translated to the KCPM. The author of this paper concludes that the KCPM is not formal – nothing is told about formalism in [17] and [18]. Moreover, the mapping to a class diagram is not strict. The mapping rules are divided into laws and proposals; the designer may accept the proposal or take another decision [18]. Hence, there is no formal transformation to a class diagram.

In the overviewed approaches, the CIM is created informally. Hence these approaches do not share benefits of formal domain modeling (mentioned in Section I). Since the CIM is informal, it is hard to define a formal transformation from the CIM to the PIM – an unambiguous transformation that can be automated. TFM, in its turn, is a formal CIM and the formal transformation to the PIM is defined.

III. Topological Functioning Model for Model Driven Architecture Approach

Nowadays an object-oriented approach is most widely used in software development. In object-oriented approaches, for example, Rational Unified Process (RUP) [19], the problem domain is not modeled formally, and the development is commonly driven by use case modeling. This tendency is disputable, because a use case diagram is fragmentary. There is no way to determine whether a created use case diagram is complete or something is missing. This also refers to the list of requirements for the software system. Furthermore, only a proper problem domain model provides a powerful language for expressing requirements for the system [20]. Explicit problem domain model gives an opportunity to understand how the system (e.g., business system) is working without software which is planned to be developed, and how this system will be influenced by the software. This way it is possible to understand not only what the clients want, but also what they need – so records are added to the list of requirements. If the client’s needs and desires are clearly determined, the probability of their satisfaction with software product essentially increases. A proper model is a formal model. Hence, the formalism must be involved in the very early stage of software development [20].

Model Driven Architecture (MDA) is an approach to system development, which increases the power of models in this study. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification [1]. Model transformation forms a key part of MDA.

CIM is a Computation Independent Model, PIM is a Platform Independent Model, and PSM is a Platform Specific Model. With the help of model transformations, going by the path CIM $\rightarrow$ PIM $\rightarrow$ PSM $\rightarrow$ software code, from an abstract model (CIM) a detailed model (PSM) is obtained. It is possible to generate a software source code from the PSM.

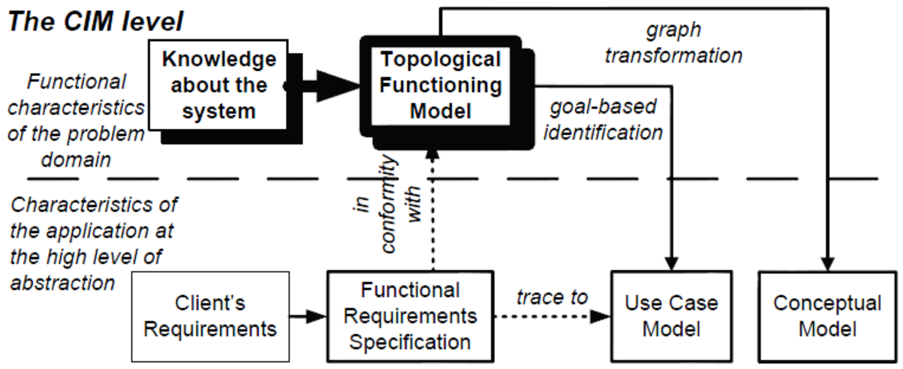
The CIM level
Functional characteristics of the problem domain
Knowledge about the system
Topological Functioning Model
graph transformation
goal-based identification
in conformity with
Characteristics of the application at the high level of abstraction
Client's Requirements
Functional Requirements Specification
trace to
Use Case Model
Conceptual Model

Fig. 1. CIM creation with the TFM4MDA (taken from [2]).

The requirements for the system are modeled in a Computation Independent Model, CIM describing the situation in which the system will be used. Such a model is called a domain model or a business model [21]. It may hide much or all information about the use of automated data processing systems. Typically such a model is independent of how the system is implemented. A CIM is a model of a system that shows the system in the environment in which it will operate, and, thus, it helps in presenting exactly what the system is expected to do. Topological Functioning Model has the above-mentioned characteristics of CIM.

Topological Functioning Model is a formal model that describes the functioning of system. The TFM has a solid mathematical base. It is represented in the form of a topological space (X, Θ), where X is a finite set of functional features of the system under consideration, and Θ is topology that satisfies axioms of topological structures and is represented in the form of a directed graph [22]. The TFM functional features describe the system physical or biological characteristics that are relevant for the normal functioning of the system. The TFM topology consists of cause-effect relations between functional features. Cause-effect relation exists between two functional features, if appearance of one functional feature is caused by appearance of the other without participation of any middle functional feature [22]. Cause-effect relations form causal chains. Causal chains must form at least one functioning cycle within the TFM. All the cycles and subcycles should be carefully analyzed in order to completely identify existing functionality of the system. The main cycle (cycles) of system functioning (i.e., functionality that is vitally necessary for system life) must be found and analyzed before starting a further analysis. TFM has topological (*connectedness*, *closure*, *neighborhood*, and *continuous mapping*) and functional (*cause-effect relations*, *cycle structure*, *inputs* and *outputs*) characteristics. Due to topological and functional characteristics mentioned above, the TFM comprises two aspects of the system – both structural and behavioral [4].

It is proposed to use the TFM as a formal CIM in the framework of MDA to model the problem domain [4]. This approach is called *Topological Functioning Modeling for Model Driven Architecture* (TFM4MDA) [2]. TFM4MDA is a model-driven approach that is based on the formalism of TFM. Fig. 1 illustrates the place of CIM (which is the TFM) in the approach.

There are two stages of the problem analysis: analysis of the problem domain and analysis of the application (solution) domain. These levels should be analyzed separately. TFM considers problem domain information separate from the application domain information captured in requirements and, thus, satisfies the main principle of MDA – separation of views [23]. The horizontal dashed line in Fig. 1 separates the problem domain (above) from the application domain (below). The knowledge about the problem domain is entered into the TFM and the TFM "as is" is developed [24]. The requirements are mapped onto the TFM functional features, so the requirements are validated and the TFM is modified. In this way, the TFM "to be" is developed – a model of problem domain which will be supported by required software [25]. It is possible to create a use case model [26] and a conceptual class model from the TFM. Mapping requirements onto functional features and creation of use case model and conceptual class model are described in detail in [4], [27].

TFM of a complex technical or business system can be constructed from its informal verbal description – the formal method is described in detail in [4], which is based on [28]. Another approach for TFM creation is the Integrated Domain Modeling approach (IDM). By using the IDM approach, knowledge about a problem domain is represented by ontology and business use cases [29]. Ontology represents the declarative knowledge (structure), and business use cases represent the procedural knowledge (behavior) about the system. Business use cases must be in conformity with ontology – verification takes place, and the models are modified until the conformity is achieved. Then the TFM can be created from business use cases. The construction of TFM from business use cases can be done automatically by using the tool [29].

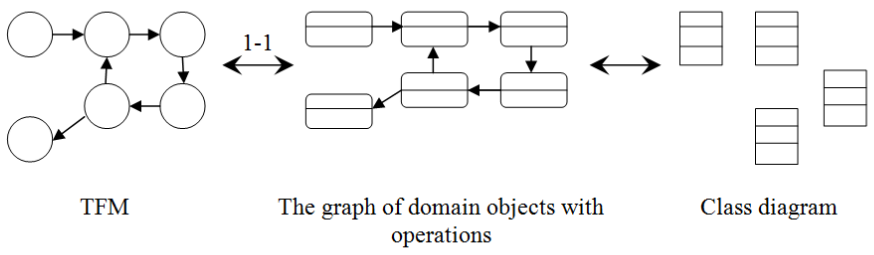

Fig. 2. The process of getting a class diagram from the TFM.

IV. Getting a UML Class Diagram from the TFM

The goal of software development is to get the software source code. As mentioned before, to get the source code (within thw MDA) we need to go by the path CIM → PIM → PSM → source code. Thus, in the beginning the PIM must be created from the CIM. UML class diagram [30] can serve as PIM which represents the structure of a system. Class diagram can be detailed to the PSM level, although it is a task of the future research. This paper focuses on the construction of a UML class diagram on the PIM level from the TFM (TFM is a CIM).

The approach of construction of topological UML class diagram from the TFM is described in [31]. Topological class diagram has topological relationships (see Section IV. B). There is no algorithm for automatic transformation from TFM to a topological class diagram.

As mentioned before, the TFM consists of a set of functional features and cause-effect relations between functional features.

A. TFM Functional Features

Within the TFM4MDA each functional feature is a 5-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 catalog etc.), *PrCond* is a set *PrCond* = $\{c_1, \ldots, c_i\}$, where c_i is a precondition or an atomic business rule (it is an optional parameter), and *E* is an entity responsible for performing actions [4]. In [31] attributes are added, forming the 8-tuple: <*A*, *R*, *O*, *PrCond*, ***PostCond***, *E*, ***Cl***, ***Op***>, where *PostCond* is a set *PostCond* = {*p*1, …, *pi*}, where pi is a postcondition or an atomic business rule; *Cl* – *Class* – is a class which will represent the object in a system static (structure) model and which will contain an operation for functionality defined by this functional feature; *Op* – *Operation* – is an operation which will contain functionality defined by a functional feature. The main idea is that the functionality of each functional feature must be realized by an individual class method. Thus, *Cl* and *Op* attributes are needed to construct a class diagram from the TFM: *Cl* is a name of a class, and *Op* is a name of a method. *Cl* and *Op* attributes are initialized (values are assigned) only when a class diagram is needed to be constructed. Other 8-tuple attributes (apart from *Cl* and *Op*) are not displayed in a class diagram; however, they help to initialize *Cl* and *Op* attributes.

B. TFM Topology

UML specification [5] does not propose a type of relation between classes that can be compared with topological (cause-effect) relation [31]. For this reason, a topological relation between classes is introduced [31]. However, this solution requires the extension of meta-model of class diagram with the goal to create the meta-model of topological class diagram, which has the description of topological relations [32]. Modifying the meta-model is bad because of the following reasons: many software tools are constructed based on the standard UML meta-model and are not able to work with other meta-models [30]; there is a possibility that a user (e.g., a system architect) would not like to work with the class diagram which differs from the standard one. For these reasons, we focus on the transformation from the TFM to the standard UML class diagram. Since TFM cause-effect relations cannot be transformed to any UML standard relation between classes, the author suggests that the class diagram, which is a result of transformation from the TFM, has no relations. Relations are added during the refinement of the obtained class diagram [33].

C. Transformation from the TFM to a Class Diagram

To execute the transformation from the TFM to a UML class diagram TFM, the attributes *Cl* and *Op* of functional features must be initialized (not necessary all of them). It is a user's (e.g., system architect's) responsibility.

In order to obtain a class diagram, first of all a graph of problem domain objects must be developed from the TFM. It is a simple transformation, where all unnecessary attributes of TFM functional features are cut – only *Cl* and *Op* remain. Then the graph vertices with similar *Cl* values are merged and a new class is created – with name *Cl* – and the class list of methods consists of *Op* values of these vertices [31]. Fig. 2 shows the process of creating the class diagram from the TFM.

D. Introducing the Automation

The author proposes automating the process part which starts after assigning values to *Cl* and *Op* attributes (this is done manually). In [31] and [34] there are no guidelines and the way of creating *Cl* and *Op* values is not clear. Thus, the development of guidelines for initializing *Cl* and *Op* requires the future research. The transformation ends with creation of the class diagram.

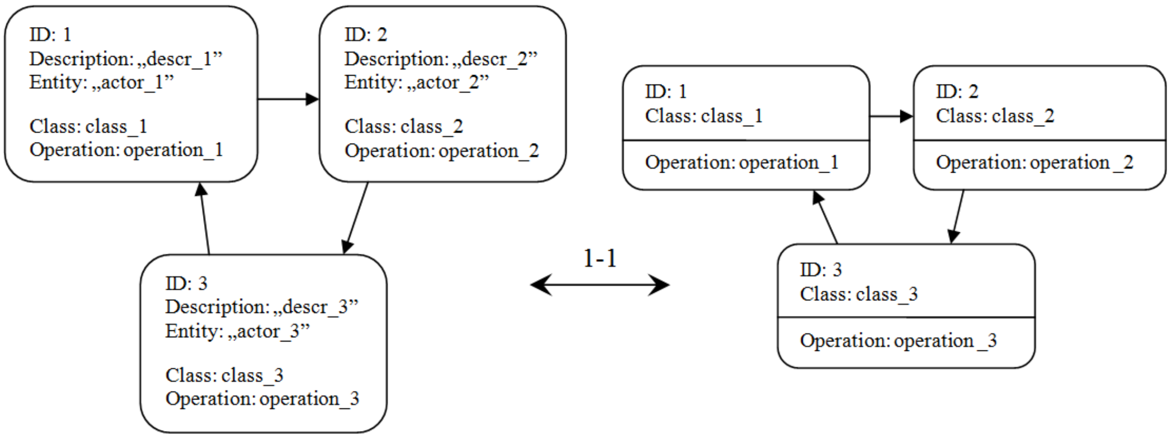

Fig. 3. An example of developing a graph of problem domain objects from the TFM.

Since the graph of domain objects with operations serves as a linking model, the author proposes not displaying this model, but only creating it in memory during execution of the transformation program. As a result of the automated transformation, the *initial* class diagram on the PIM level is created. This diagram consists of classes with names and lists of methods. The refinement of the initial class diagram is done manually [33].

The automation of model transformation facilitates user's (analyst, system architect). Therefore, the cost of software development is decreased. This way the system analysis stage (TFM development) is related to the development of UML model on the PIM level.

V. The Transformation Algorithm for Getting a UML Class Diagram from the TFM

A. Getting a Graph of Problem Domain Objects from the TFM

Firstly, the graph of problem domain objects with operations must be developed from the TFM. For each TFM functional feature, a vertex in the graph must be created and its attributes must be initialized with the corresponding functional feature attributes. Fig. 3 shows an example of developing the graph of problem domain objects from the TFM. Attribute ID (identifier) is added for algorithm realization. Attribute *Description* consists of the following functional feature attributes: action (*A*); result (*R*); object (*O*) (Section IV. A).

The algorithm for developing the graph of problem domain objects from the TFM in a pseudocode:

```
// The vertex of the problem domain
// object graph is described by the
// following code:
struct DomainObjectVertex
{
  id : Integer;  // primary key
  class : String;
  operation : String;

  // The set of integer numbers which includes
  // identifiers of vertices which are connected to
  // the given vertex with an oriented edge.
  // The edge is oriented from the given (this)
  // vertex to the vertex, which identifier is
  // included in the set.
  edges : Set of Integer;
};

// The TFM's functional feature is described by the
// following code:
struct FunctionalFeature
{
  id : Integer;  // primary key
  description : String;
  entity : String;
  class : String;  // Cl attribute
  operation : String;  // Op attribute
};

// Topological (cause-effect) relationship is
// described by the following code:
struct TopologicalRelationship
{
  // id of "cause" functional feature:
  source : Integer;
  // id of "effect" functional feature:
  target : Integer;
};

T: is a set of TFM's functionalfeatures;
 t[i] is a functional feature with id = i;
G: is a set of vertexes of the problem
 domain object graph;
 g[i] is a vertex with id = i;
R: is a set of topological relationships;

At the beginning:
{
  G = Ø (empty set);
  T includes all TFM's functional features;
  R includes all topological relationships from TFM.
}

// The problem domain object graph is developed
// iteratively. During iteration a vertex is
// created and added into the set G.
// T.size() - number of functional features in
// the set T.
For i:=1 to T.size() do
```

```
{
  // create new vertex of object graph:
  create DomainObjectVerticy type variable v;
  v.id := i;
  v.class := t[i].class;
  v.operation := t[i].operation;

  // the set of edges will be created
  // later, for now it is an empty set:
  v.edges := Ø;

  // add vertex v into the set G:
  G := G ∪ {v};
}

// declaration of variable r:
r - TopologicalRelationship type instance;

// Transferring of TFM relationships into the
// object graph. Process runs iteratively.
// During iteration r becomes an element of
// the set R.
// r.source is a "cause" functional feature's id
// and also the corresponding vertex's id.
// Hence g[r.source] is graph's vertex from which
// the edge comes out.
// r.target is the object graph's vertex into
// which the edge under consideration incomes.
// Hence r.target value must be added
// into the g[r.source].edges set.
For all r ∈ R do
  g[r.source].edges :=
    g[r.source].edges ∪ {r.target};
```

B. Getting a UML Class Diagram from the Constructed Graph of Problem Domain Objects

The attributes *class* and *operation* of vertices in the developed graph of problem domain objects are equal to the attributes *Cl* and *Op* of TFM functional features that correspond to these vertices. If *Cl* or *Op* attribute of a functional feature is empty, then the corresponding attribute of the corresponding vertex in the graph is also empty. For this reason, a user (e.g., a system architect) has an opportunity to check the class diagram before assigning values to *all Cl* and *Op* attributes in the TFM. Hence, the algorithm must support the creation of the class diagram from the TFM in which *not all Cl* and *Op* attributes are initialized (the value is assigned). Four cases are possible:

1) Both *Cl* and *Op* attributes of a functional feature are initialized. In this case, the corresponding vertex of the graph participates in construction of the class diagram – both class name and operation name are taken into account.

2) *Cl* attribute is initialized, but *Op* – is not. In this case, the vertex does not add a new operation, but the class with the name equal to a value of *class* attribute is added to the class diagram.

3) *Op* attribute is initialized, but *Cl* – is not. In this case, the vertex cannot participate in construction of the class diagram, and the value of its *operation* attribute is lost (it stays in the TFM, but it is not transferred to the class diagram).

4) Neither *Cl* nor *Op* attribute is initialized. In this case, the vertex is treated in a similar way to the third case.

It is possible to create the class diagram from the constructed graph of problem domain objects. The vertices of the graph with the same type of objects (*class* values) must be merged [35]. Since it is not possible to transform the relationships between TFM functional features to the class diagram (Section IV. B), the edges of the graph are lost.

Class attributes (in the class diagram) are generated from getter and setter methods (whose names start with *get* or *set*). Corresponding method is retained in the list of methods of the class despite the fact that the existence of an attribute implicitly indicates that a corresponding setter and getter exist. The method needs to be there so that a user (e.g., a system architect) could see that the attribute was generated from a method that was transformed from the TFM.

The algorithm of creating a UML class diagram from the graph of problem domain objects in a pseudocode:

```
// The class of UML class diagram is
// described by the following code:
struct Class
{
  className : String;
  // list of attributes:
  attributes : List of String;
  // list of methods:
  operations : List of String;
};

G: is a set of vertexes of the problem
 domain object graph; g[i] is a vertex
 with id = i;
C: is a set of UML classes;
 c is an element of the set C (a class);
At the beginning:
{
  C = Ø (empty set);
  the set G was developed;
}

// The set C is developed iteratively.
// During iteration one element of the set G
// (one vertex) is inspected.
// The information that includes the vertex is
// used to develop the set C.
// G.size() - the number of vertices in the set G.
For i:=1 to G.size() do
{
  // Firstly, the attribute class is checked.
  // If it is empty, then the vertex
  // does not improve the set C.
  IF g[i].class is not empty, THEN
  {
    // Then the set C is checked whether it has
    // an element with a class name equal to
    // vertex's g[i] class attribute.
    // If it does not have, then a new class
    // is added into the set C.
    IF C does not have a class with
     className that is equal to g[i].class, THEN
    {
      // create a new class:
      create Class type variable cNew;
      cNew.className := g[i].class;
      // for now lists of attributes
      // and methods are empty:
```

```
      cNew.attributes := Ø;
      cNew.operations := Ø;
      // add the class cNew into set C:
      C := C U {cNew};
    }

    Designation: cCurrent - the C set's class which
     attribute className is equal to g[i].class;

    // The operation attribute of vertex g[i]
    // is checked. If it is not empty, then
    // cCurrent.operations list is checked
    // whether it has an element that is equal to
    // g[i].operation. If there is no such method
    // in the list, then it is added.
    IF g[i].operation is not empty,
     THEN
      IF g[i].operation is not in the
       list cCurrent.operations, THEN
        cCurrent.operations :=
          cCurrent.operations U{g[i].operation};
  }
  // Here ends the code block, which is executed
  // if condition "IF g[i].class is not empty"
  // is met.
}
// The "For i:=1 to G.size() do" loop ends here.

// declaration of variable c:
c - Class type instance;
// declaration of variable oper:
oper -String type instance;

// Generation of class's attributes.
// The set C is processed iteratively.
// During iteration one class is inspected.
For all c ∈ C
{
  // Each method of a class is analyzed in turn.
  For all oper ∈ c.operations do
  {
    IF oper begins with „set" or with „Set", or
     with „get", or with „Get", THEN
    {
      create String type variable newAttribute;
      newAttribute := oper;

      // To obtain the corresponding name of
      // attribute the word „set" or „get" is cut.
      crop the first 3 symbols of newAttribute;
      // Brackets are also cut.
      IF last two symbols of
       newAttribute are „()", THEN
        crop the last 2 symbols of newAttribute;

      // Attribute's first letter should be written
      // in lower case.
      IF the first symbol of
       newAttribute is written in upper case, THEN
        replace the first letter of newAttribute
         with the corresponding lower case letter;
```

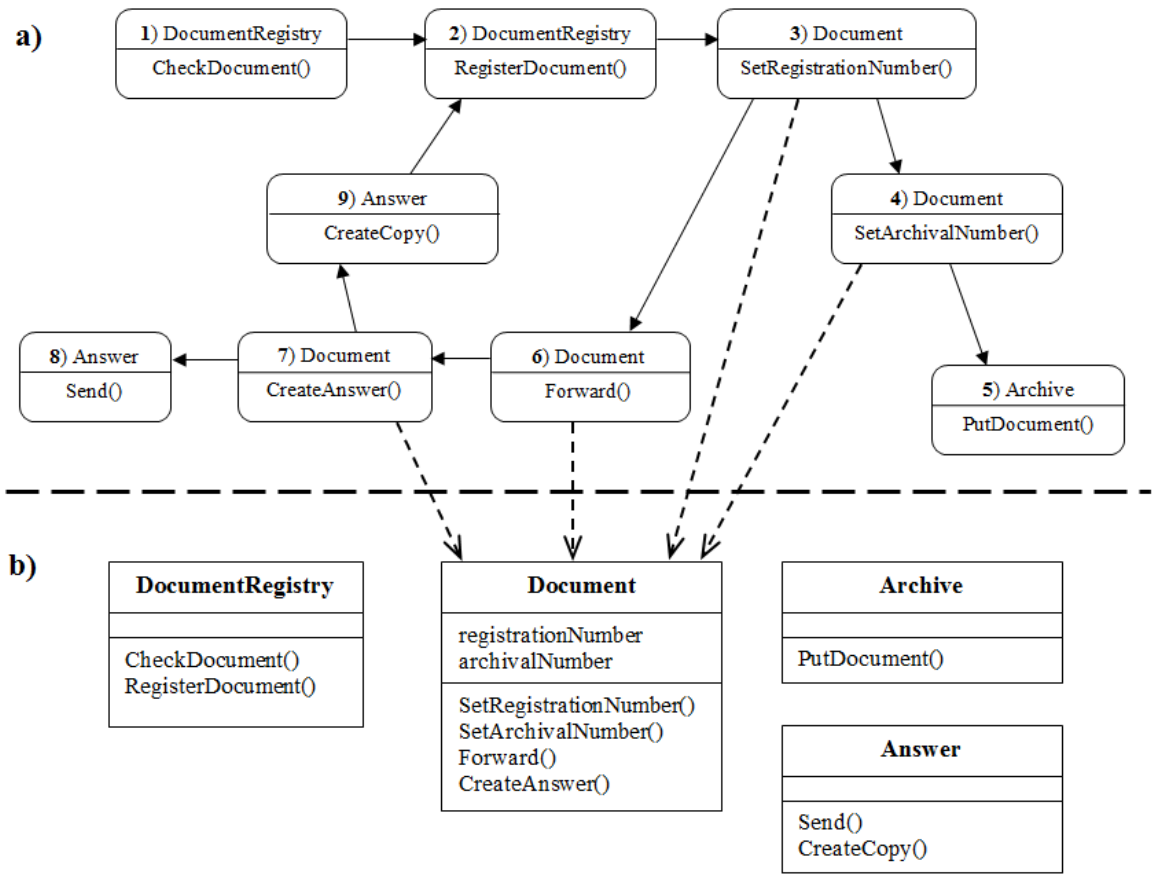

Fig. 4. Example of transforming a graph of problem domain objects (a) to a UML class diagram (b) – result of algorithm execution.

```
      // Before adding newAttribute into the list
      // of attributes we need to check if the list
      // does not already have an attribute with
      // the same name.
      IF newAttribute is not in the
       list c.attributes, THEN
        c.attributes :=
          c.attributes ∪ {newAttribute};
    }
  }
  // The „For all oper ∈ c.operations do”
  // loop ends here.
}
// The „For all c ∈ C do” loop ends here

// After executing the above mentioned algorithm
// the set C is ready to be used for the class
// diagram construction. Classes are transferred to
// the UML class diagram space.
For all c ∈ C do
{
  place a new class in the UML class
   diagram and mark it as cDiagram;
  assign cDiagram the class name c.className;
  add to the list of attributes of cDiagram
   all attributes from the list c.attributes;
  add to the list of methods of cDiagram all
   methods from the list c.operations;
}
```

Fig. 4 shows an example of getting a UML class diagram (b) from a graph of problem domain objects (a). The dashed arrows show that the objects with the same object type "Document" are used to create a class with the same name. The attributes of the class are generated from getter and setter methods.

As a result of the transformation, the *initial* UML class diagram on the PIM level is created (with attributes and operations). To obtain the complete class diagram on the PIM level, the initial class diagram must be refined [33]. The refinement of a class diagram is aimed to lower an abstraction level of it. By lowering an abstraction level, the diagram gets additional information, which is needed during the software development and later during its maintenance.

VI. Conclusion

This research focused on creation of a UML class diagram from a Topological Functioning Model. The author worked on decreasing the costs of software development within the TFM4MDA approach, which was related to creation of a UML class diagram on the PIM level from the TFM on the CIM level. The decrease can be achieved by automating the formal transformation from the TFM to a class diagram. The main accomplishment of this study is the developed algorithm of transformation from the TFM to an initial UML class diagram on the PIM level. The algorithm is written in a pseudocode. It can be implemented as a tool, thus improving the TFM4MDA approach. Thus, the link between the beginning stage of system analysis (the development of TFM) and the development of PIM becomes stronger.

The next task is to implement the introduced transformation algorithm as a tool. Thus, the TFM4MDA approach will become more efficient. To practically validate the result of the work, a tool (or tool prototype) must be developed. Theoretically, working with a tool that executes the transformation is more effective than manually creating the initial class diagram (classes with operations). First of all, the larger the TFM is, the harder it becomes for manual processing. The probability of making mistakes grows. The automatic transformation nullifies the risk of making mistakes during the transformation. Secondly, the user must initialize *Cl* and *Op* attributes only once for each functional feature. During the development process, the TFM will most likely be modified at least several times. After a modification, the retained functional features will still have the initialized *Cl* and *Op* attributes, which will be used for the creation of a class diagram. This approach is more effective than manually recreating a class diagram, or trying to modify it accordingly to the new version of TFM. Thirdly, working directly with the TFM in the TFM editor would be more comfortable than working with the TFM and a UML class diagram in two different editors during manual transformation.

It is not yet known how the changes in the class diagram should affect the TFM and whether they should affect the TFM. It would be better if the modifications in the TFM affected the class diagram. In this case, the user would not have to start from the initial class diagram after modifying the TFM. For now the developed transformation algorithm only creates a new initial class diagram that conforms to the TFM. The solutions for these problems should be found in the future research.

References

[1] Miller, J. and Mukerji, J., "MDA Guide Version 1.0.1", OMG. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01. [Accessed: 14 March 2015].

[2] Osis, J., Asnina, E. and Grave, A., "Formal Computation Independent Model of the Problem Domain within the MDA," in *Information Systems and Formal Models, Proc. of the 10th Int. Conf., ISIM'07*, Silesian University in Opava, Czech Republic, 2007, pp. 47–54.

[3] Osis, J., Asnina, E. and Grave, A., "Computation Independent Modeling within the MDA," in *Proc. of the IEEE Int. Conf. on Software Science, Technology and Engineering*, Oct. 30–31, 2007, Herzlia, Israel, IEEE Computer Society Nr. E3021, pp. 22–34. http://dx.doi.org/10.1109/SwSTE.2007.20

[4] Osis, J. and Asnina, E., "Topological Modeling for Model-Driven Domain Analysis and Software Development: Functions and Architectures," in *Model-Driven Domain Analysis and Software Development: Architectures and Functions*. IGI Global, Hershey – New York, 2011, pp. 15–39. http://dx.doi.org/10.4018/978-1-61692-874-2.ch002

[5] OMG (Object Management Group), "OMG Unified Modeling Language TM (OMG UML), Superstructure, Version 2.4.1.", 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/. [Accessed: 14 March 2015].

[6] OMG (Object Management Group), "Business Process Model and Notation (BPMN), Version 2.0.2.", 2013. [Online]. Available: www.omg.org/spec/BPMN/2.0.2/PDF. [Accessed: 14 March 2015].

[7] Linagora, "What is MDA? Why concerns BPMN?" [Online]. Available: https://research.linagora.com/pages/viewpage.action?pageId=3639295. [Accessed: 14 March 2014].

[8] Bao, N. Q., "A proposal for a method to translate BPMN model into UML activity diagram," Vietnamese-German University – BIS, 2010. [Online]. Available: http://www.nqbao.com/archives/files/BPMN-UMLAD.pdf. [Accessed: 14 March 2014].

[9] The Open Group, "ArchiMate 2.1 Specification", 2012-2013. [Online]. Available: http://pubs.opengroup.org/architecture/archimate2-doc/toc.html. [Accessed: 14 March 2014].

[10] OMG (Object Management Group), "OMG Meta Object Facility (MOF) Core Specification, Version 2.4.2.", 2014. [Online]. Available: http://www.omg.org/spec/MOF/2.4.2/PDF/. [Accessed: 14 March 2015].

[11] Armstrong, C., Baker, J.D., Band, I., *et al*., "Using the ArchiMate® Language with UML®", 2013. [Online]. Available: http://cdn2.hubspot.net/hub/183807/file-1805596253-pdf/site/media/downloads/W134.pdf?t=1418385713847. [Accessed: 14 March 2014].

[12] Liu, D., "Automating Transition from Use Cases to Class Mode", Master Thesis. Calgary: University of Calgary, 2003.

[13] Liu, D., Subramaniam, K., Eberlein, A., Far, B.H., "Natural Language Requirements Analysis and Class Model Generation Using UCDA" in *Innovations in Applied Artificial Intelligence: 17th Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*. Berlin : Springer, 2004, pp. 295–304.

[14] Overmyer, S. P., Benoit, L., Rambow, O., "Conceptual Modeling through Linguistic Analysis Using LIDA," *Software Engineering*, 2001, pp. 401–410. http://dx.doi.org/10.1109/icse.2001.919113

[15] OMG (Object Management Group), "Semantics of Business Vocabulary and Business Rules (SBVR), Version 1.2.", 2013. [Online]. Available: http://www.omg.org/spec/SBVR/1.2/PDF/. [Accessed: 14 March 2015].

[16] Raj, A., Prabhakar, T.V., Hendryx, S., "Transformation of SBVR Business Design to UML Models" in *ISEC '08 Proc. of the 1st India software engineering conference*, Hyderabad, India, Feb. 19–22, 2008, pp. 29–38. ISBN: 978-1-59593-917-3. http://dx.doi.org/10.1145/1342211.1342221

[17] Fliedl, G., Kop, C. and Mayr, H.C., *et al*., "Deriving static and dynamic concepts from software requirements using sophisticated tagging," *Data & Knowledge Engineering*, 2007, pp. 433–448. http://dx.doi.org/10.1016/j.datak.2006.06.012

[18] Mayr, H.C. and Kop, Ch., "A user centered approach to requirements modelling" in *Proc. Modellierung 2002*, Lecture Notes in Informatics LNI p-12, GI-Edition, 2002, pp. 75–86.

[19] Rational, "Rational Unified Process. Best Practices for Software Development Teams". [Online]. Available: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf. [Accessed: 14 March 2015].

[20] Osis, J. and Asnina, E., "Is Modeling a Treatment for the Weakness of Software Engineering?" in *Model-Driven Domain Analysis and Software Development: Architectures and Functions*. IGI Global, Hershey - New York, 2011, pp. 1–14. http://dx.doi.org/10.4018/978-1-61692-874-2.ch001

[21] Osis, J. and Asnina, E., "Topological Functioning Model as a CIM-Business Model" in: *Model-Driven Domain Analysis and Software Development: Architectures and Functions*. IGI Global, Hershey – New York, 2011, pp. 40–64. http://dx.doi.org/10.4018/978-1-61692-874-2.ch003

[22] Osis, J., "Topological Model of System Functioning" (in Russian) in *Automatics and Computer Science, J. of Academia of Siences, Riga, Latvia*, no. 6, 1969, pp. 44–50.

[23] Asnina, E. and Osis, J., "Computation Independent Models: Bridging Problem and Solution Domains" in J. Osis, O. Nikiforova (Eds.). *Model-Driven Architecture and Modeling Theory-Driven Development: ENASE 2010*, 2ndMDA&MTDD Whs., SciTePress, Portugal, 2010, pp. 23–32.

[24] Osis, J. and Asnina, E., *Model-Driven Domain Analysis and Software Development: Architectures and Functions*. IGI Global, Hershey – New York, 2011, 487 p. http://dx.doi.org/10.4018/978-1-61692-874-2

[25] Osis, J. and Asnina, E., "A Business Model to Make Software Development Less Intuitive," *Proc. of the 2008 Int.Conf. on Innovation in Software Engineering*, Vienna, Austria. IEEE Computer Society CPS, Los Alamitos, USA, 2008, pp. 1240–1246.

[26] Osis, J. and Asnina, E., "Derivation of Use Cases from the Topological Computation Independent Business Model" in *Model-Driven Domain Analysis and Software Development: Architectures and Functions*. IGI Global, Hershey – New York, 2011, pp. 65–89. http://dx.doi.org/10.4018/978-1-61692-874-2.ch004

[27] Osis, J., Asnina, E. and Grave, A., *MDA* Oriented Computation Independent Modeling of the Problem Domain. *Proceedings of the 2nd International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2007)*, Barcelona, Spain, 2007, pp. 66–71.

[28] Booch, G., *Object-Oriented Design with Applications*. Addison Wesley Longman, Inc, 1994.

[29] Slihte, A., Osis, J. and Donins, U., "Knowledge Integration for Domain Modeling," *Proc. of the 3rd Int. Workshop on Model-Driven Architecture and Modeling-Driven Software Development*, China, Beijing, 8–11 June, 2011. Lisbon: SciTePress, 2011, pp. 46–56. ISBN 9789898425591.

[30] Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*. 2nd ed. Addison-Wesley, Reading, 2004, 721 p. ISBN 978-0321245625.

[31] Osis, J. and Donins, U., "Formalization of the UML Class Diagrams," *Evaluation of Novel Approaches to Software Engineering: 3rd and 4th Int. Conf. ENASE 2008/2009: Revised Selected Papers*, Italy, Milan, 9–10 May, 2010. Berlin: Springer-Verlag, 2010, pp. 180–192. ISBN 9783642148187. E-ISBN 9783642148194. ISSN 1865-0929

[32] Osis, J. and Donins, U., "Platform Independent Model Development by Means of Topological Class Diagrams," in *5th Int. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE 2010) / Model-Driven Architecture and Modeling Theory-Driven Development*. Greece, Athens, July 22–24, 2010. Portugal: SciTePress, 2010, pp. 13–22. ISBN 9789898425164.

[33] Donins, U., Osis, J., Slihte, A., Asnina, E. and Gulbis, B., "Towards the Refinement of Topological Class Diagram as a Platform Independent Model," in J. Osis, O. Nikiforova (Eds.). *Model-Driven Architecture and Modeling-Driven Software Development: ENASE 2011, 3rd Whs.* MDA&MDSD, SciTePress, Portugal, 2011, pp. 79–88.

[34] Donins, U., "Software Development with the Emphasis on Topology" in *Advances in Databases and Information Systems: Lecture Notes in Computer Science*. vol. 5968. Berlin: Springer Berlin Heidelberg, 2010, pp. 220–228. ISBN 9783642120817. E-ISBN 9783642120824. ISSN 0302-9743.

[35] Osis, J., Asnina, E. and Grave, A., "Formal Problem Domain Modeling within MDA", *Communications in Computer and Information Science*, CCIS, vol. 22, Software and Data Technologies, Springer-Verlag Berlin Heidelberg, 2008, pp. 387–398.

Arturs Solomencevs obtained the Bachelor Degree in Computer Control and Computer Science from Riga Technical University, Latvia, in 2014.

Currently he is the first-year Master Student and Scientific Assistant at the Department of Applied Computer Science, Riga Technical University. He actively participates in the scientific research project called Topological Functioning Model for Software Engineering (TFM4SE).

E-mail: Arturs.Solomencevs@gmail.com