Applied Computer Systems doi: 10.1515/acss-2015-0018

2015/18

Obtaining and Visualization of the Topological Functioning Model from the UML Model

Viktoria Ovchinnikova
Riga Technical University, Latvia

Abstract **– A domain model can provide compact information about its corresponding software system for business people. If the software system exists without its domain model and documentation it is time-consuming to understand its behavior and structure only from the code. Reverse Engineering (RE) tools can be used for obtaining behavior and structure of the software system from source code. After that the domain model can be created. A short overview and an example of obtaining the domain model, Topological Functioning Model (TFM), from source code are provided in the paper. Positive and negative effects of the process of TFM backward derivation are also discussed.**

Keywords **– Domain model, model-driven architecture, reverse engineering, topological functioning model, UML diagrams.**

I. Introduction

It is difficult to know the software system by its source code without any system skeleton (model) and documentation, because each developer has his own style of and guidelines for programming. Therefore, development and maintenance of such software systems is a time-consuming and costly process.

The backward obtaining of the domain model can be used in such situations, e.g., when migrating to a new technology platform or integrating a new software system. It is used when the information about the software system behavior and structure is absent. . As well as when there is a deficiency of resources can also be used for ordering the effective and qualitative development of software system. Legacy software system architecture is more understandable when it is visualized as a graphical model.

The software system can be displayed graphically at the high level of abstraction as Topological Functioning Model (TFM) [9], Business Process Model and Notation (BPMN) [21], i star (i*) [22], e3 value [23], ArchiMate [24], Event-driven Process Chain (EPC) [25] and others. This paper focuses mainly on TFM. It is the main domain model (formal and mathematical) in the software development method Topological Functioning Modeling for Model-Driven Architecture (TFM4MDA) [10]. The system will be displayed as an oriented graph, wherein vertices with names (denote functioning) and relationships between them exist. The system is fully overviewed (all functional characteristics of the system are provided and all of them are connected) by the TFM. Functioning of real-world system is continuous and TFM provides this continuity by the main cycle. Various input and output signals are necessary for real-world system to function and can affect its core functionality – TFM provides these inputs and outputs.

Obtainment of the domain model from the source code is not a fully automated process. MDRE (Model-driven Reverse Engineering) approach provides obtainment of models (Unified Modeling Language (UML) diagrams) from legacy source code. The author of [26] considers that all necessary information of legacy software system can be taken from the source code. The model derived from the source code can be expressed by using Object Constraint Language (OCL) and UML [26], [27]. ADMTF (Architecture Driven Modernization Task Force) uses model transformation for taking the legacy software system more agile [28]. The ADMTF main standard is Knowledge Discovery Meta-model (KDM). The authors of [29] consider that ADMTF processes are transformation between models in such a way from legacy source code to KDM to BPMN and Semantic of Business Vocabulary and Rules (SBVR) to upgraded BPMN and SBVR to UML diagrams to generated source code. In our case both approaches are appropriate, but in this paper we focus only on TFM and UML diagrams, because many different tools exist for UML derivation from the legacy source code. The obtainment of UML diagrams from the TFM is considered in [4], [5], [11]. Some rules mentioned there can be used as backward obtainment of TFM from the UML diagrams.

Reverse Engineering (RE) is necessary for examination and analysis of legacy software system to provide behavior and structure of the system in our research. The TFM “as is” can be obtained from the UML model and TFM “to be” can be supplemented during the analysis of TFM “as is” and new requirements for the system. As the result the “target” system can be obtained. The transformation algorithm was theoretically developed in [1] and [2] by the mappings between TFM and UML model elements was provided in [18]. It is necessary to implement this algorithm, validate and check it by practical experiment with already developed legacy system, because some information can be lost during transformations (e.g. relationships between elements). The connectedness of all elements needs to be checked in the obtained TFM. The main problem can be unclear names provided in the source code.

In our case the obtainment of the domain model from the source code can be divided into two parts. First, structure and behavior of the software system can be automatically generated as a model from the source code using RE tools. After that the TFM can be obtained from the generated model using the partially implemented transformation algorithm [1], [2].

The goal of this paper is to discuss the results of obtaining the TFM from the source code using RE and Model-Driven Architecture (MDA) principles. Discussion of the positive and negative effects of the transformation is also provided.

The paper is structured as follows: Section II shortly describes TFM4MDA and RE; Section III discusses the transformation algorithm in brief; Section IV illustrates the example and results of using the transformation algorithm; Section V contains information about the positive and negative effects of the transformation process; Section VI contains conclusions and future work.

II. Topological Functioning Modeling for Model-Driven Architecture

There are two domain models – problem (the system "as is") and solution (the system "to be") domains, which are overviewed during software system modeling. The problem domain is a description of existing software system, but the solution domain – its description with included improvements. The necessary system can be obtained from the solution domain that should be involved in the problem domain. More information about domain modeling and bridging the problem and the solution domain is provided in [3], [4], [5] and [6].

MDA models show the software system at various abstract levels [7] – Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) and Implementation Specific Model (ISM). In the context of MDA the transformation between models is executed from CIM to PIM, from PIM to PSM and from PSM to ISM.

In the case of TFM4MDA, CIM is represented as TFM [8], PIM/PSM – as structure and behavior of the software system, represented by UML diagrams and ISM – as the source code. In this research, the transformation between models is backward (ISM – PIM/PSM - CIM) as it is illustrated in Fig. 1. RE techniques can be used for this transformation.

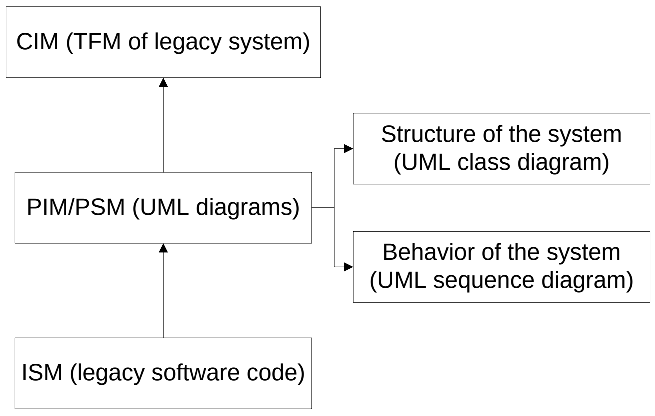

CIM (TFM of legacy system)
PIM/PSM (UML diagrams)
Structure of the system (UML class diagram)
Behavior of the system (UML sequence diagram)
ISM (legacy software code)

Fig. 1. RE within TFM4MDA.

III. The Transformation Algorithm in Brief

Functioning and topological properties [9] characterize the TFM. They provide modeling of functional characteristics of the business system. The functioning properties are cause-and-effect relations, cycle structure, inputs and outputs. And topological properties are closure, connectedness, neighborhoods and continuous mapping. Functional feature is represented as a unique structure <Id (identifier), A (object's action), R (result of the object's action), O (object), PrCond (preconditions), PostCond (post-conditions), Pr (providers), Ex (executers), Req (requirements), Cl (class), Op (operation)> [10], [11].

RE provides availability to analyze and to research the behavior and structure of the existing software system, represented at the high level of abstraction. UML diagrams can be obtained from the source code using RE techniques and tools. In our case the tool "Visual Paradigm for UML" is chosen from 13 considered tools (Imagix 4D, ArgoUML, AmaterasUML, jGRASP, Visual Paradigm for UML, Fujaba, EclipseUML, MoDisco, Apollo for Eclipse, AgileStructureViews, Diver, ObjectAid and ModelGoon) to generate UML diagrams from the source code [2], [12].

UML class diagram can represent the structure of the software system, and UML sequence diagrams – the behavior. After that these diagrams can be transformed to the TFM, using the transformation algorithm, as it is shown in Fig. 2. The obtained TFM can be represented visually. Necessary changes and supplements can be added to it by using the Integrated Domain Modeling (IDM) tool [13], [14], [15].

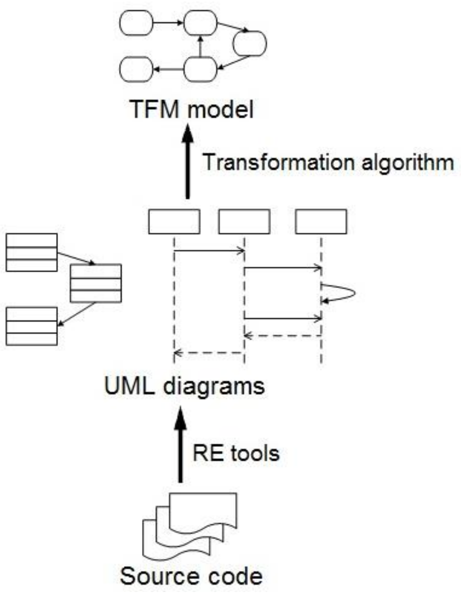


Fig. 2. The process of obtaining the TFM.

Topological UML (TopUML) modeling is the extension of UML that helps to follow cause-and-effect relations that come from the problem domain and are in the solution domain. Mappings between TFM and TopUML elements are described and examples are provided in [11], [16], [17]. Similarities and relationships between UML sequence diagrams and the TFM elements are provided in [2], [18]. From them four mapping rules are defined [2]:

The first rule: The name of TFM functional feature is similar to the name of the message from UML sequence diagrams, which can be manually defined after transformation.

TFM functional feature elements defined from the UML sequence diagrams are:

- Identifier (Id) is a unique string and can consist of message's identifier;
- Object's action (A) is the message's name till the opening bracket (without parameters, types and return types);
- Class operation (Op) is the message's name;
- Both object (O) and class (Cl) are lifelines type (class) names, to which the current message is sent;
- Preconditions (PrCond) and post-conditions (PostCond) are the frame's definition, if conditions exist within the frame;
- Executer (Ex) is defined by the type of the message, i.e., if the type of the message is an event, which is raised by a user, the executer is this user, otherwise it is the system.

The second rule: The frame definition or condition can affect the cause-and-effect relation of the TFM. The information of including data to the functional feature and of relationships of two functional features, which depends on the frame, is overviewed and provided in [2]. Elements of cause-and-effect relation are the following:

- Identifier consists of two functional feature identifiers, which are separated with semicolon;
- The cause of functional feature is the identifier of cause;
- The effect of functional feature is the identifier of effect.

The third rule: The count of TFM cycles is unlimited. It can be represented automatically after relationships of functional features have been defined. The main cycle need to be defined by the IDM user.

The fourth rule: Inputs and outputs of the TFM are messages, which go to/from an outside actor from the UML sequence diagram. They also can be defined manually after obtaining of TFM model.

The functioning properties and some functional feature elements of the TFM are considered in rules. Other functional features of the TFM cannot be defined, because it is impossible to get this information from the UML sequence diagrams. This information can be manually added during the analysis of the obtained TFM and the software system. The transformation algorithm is provided in the chart diagrams in [1] and is implemented in pseudo code in [2].

The automated part of the transformation algorithm written in QVT (Query/View/Transformation) operational language with the source code description is created following these rules and source code is shown in the next page. In this part of the transformation algorithm only functional features are obtained from the UML class diagram. This part of automated algorithm helps to ignore unnecessary operations from extended external classes, which do not provide the main logic of this legacy software system. Interfaces from the source code are not considered in this transformation algorithm, because all their operations need to be realized in the classes, which implement this interface. The TFM functional features without any relationships with its elements can be generated using this part of algorithm and the UML and the TFM meta-models. The TFM meta-model is taken from the IDM tool [15] and the UML meta-model is available on the Eclipse platform after installing UML2. These meta-models are necessary for transformation from the UML model to the TFM model.

IV. Results of Applying The Algorithm

It is necessary to generate the UML model from the source code for applying the transformation algorithm. The game "Reversi" is taken as an example of the software system. The logic of this game is described in [19]. The source code of this system is provided in [20].

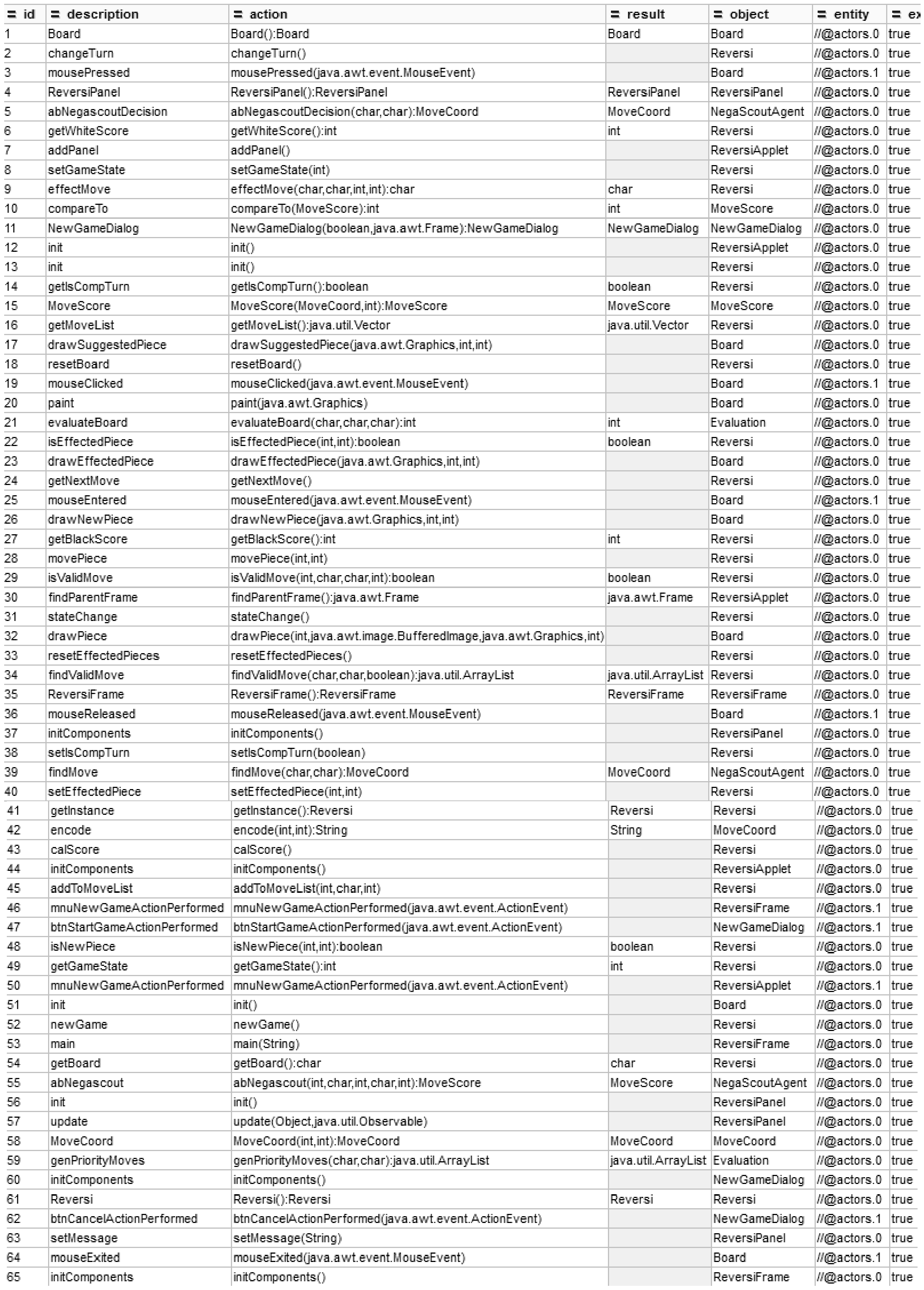
The trial version of tool "Visual Paradigm for UML" is used in order to obtain the UML diagrams. Visualizations of the UML class and sequence diagrams are discussed in [2]. It helps in understanding the source code, check correctness of it and compare necessary parts of diagrams with the source code. The results of visualization of the UML class diagrams show that there are some classes which are not related, but the exchange of message between them exists. If an object of the class is represented as a global variable of the other class, then these both classes are related. If an object of the class is represented as an internal variable in an operation of the other class, then these classes are not related. But it is not important in this research, because all relations between objects of classes are planned to be taken from the UML sequence diagrams. It is needed mainly to see the structure of the software system with included classes, their attributes and list of operations.

XML (Extensible Markup Language) metadata interchange (XMI) version 2.1 has been used to create a file with information about the UML class and sequence diagram and export it from the tool. The exported UML model, after some changes, that do not affect the logic of the software system, was used in transformation from the UML model to the TFM model. The information about access to elements and their meanings in the UML model is provided in [2].

Fig. 3 represents the information of all obtained functional features after automatic transformation from the UML model to the TFM. There are following columns in the obtained TFM model:

- id – identifier;
- Description – name of the functional feature;
- Action – full name of the operation of the UML model;
- Result – return type of the operation of the UML model;
- Object – class name from the UML model;
- Entity – executer identifier. Meta-data consists of an executer list, and entity is the identifier of the executer. Two executers exist in this system – system (//@actors.0) and user (//@actors.1);
- executerIsSystem – condition if the executer is in the system.

```
// obtaining of TFM from UML model by mappings rules
mapping UML::Model::UML2TFM() : TFM::TFM {
var user : String := "User";  // role of user outside of system
var system : String := "System";  //role of system
var actorValue : String;  // name of actor
var resultValue : String;  // return type of operation
var counter : Integer := 0;  // identifier of functional feature
var fullOperation : String;  // full operation signature
actors += map createActor(user);  // add actor to actor list
actors += map createActor(system);  // add actor to actor list
// each UML element of type PackagedElement is examined
self.packagedElement->forEach(a) {
// if it is element of type Package, then operations from class is examined and
// added to TFM
 IF (a.oclIsTypeOf(Package)) THEN {
  a.allSubobjectsOfKind(Operation)[UML::Operation]->forEach(b) {
  IF (b._class <> null and b._class.name <> '') THEN {
   fullOperation := b.name + "(";
   counter := counter + 1;
   actorValue := system;
   b.allSubobjectsOfType(Parameter)[UML::Parameter]->forEach(c) {
    IF (c.direction.toString().equalsIgnoreCase('return') and c.type <> null)
    THEN {
     resultValue := c.type.name;} ENDIF;
    IF (c.direction.toString().equalsIgnoreCase('return') and c.type = null)
    THEN {
     resultValue := c.operation.name; } ENDIF;
    IF (c.direction.toString().equalsIgnoreCase('in') and c.type <> null) THEN {
     fullOperation := fullOperation + "," + c.type.name;
     IF (c.type.name.endsWith("Event") = true) THEN {actorValue := user;} ENDIF;
    } ENDIF;
   };
   IF (fullOperation.find(",") <> 0) THEN { fullOperation :=
    fullOperation.substringBefore(",") + fullOperation.substringAfter(","); } ENDIF;
   fullOperation := fullOperation + ")";
   IF (resultValue.toString().equalsIgnoreCase("void") = false) THEN {
    fullOperation := fullOperation + ":" + resultValue;
   } ELSE { resultValue := null; } ENDIF;
    functionalFeatures += b.map operation2FunctFeat (actors,counter,null,
                               actorValue,resultValue,fullOperation);
    fullOperation := null; } ENDIF;
  };
 } ENDIF;
};
actors := actors->sortedBy(description);  // sorting of actors by names
}
mapping createActor(name : String):TFM::Actor{description := name;}// create TFM actor
mapping UML::Operation::operation2FunctFeat(actors : Set(TFM::Actor), counter :
Integer, cond : String, aName : String, rName : String, fullOperation : String) :
TFM::FunctionalFeature { // create TFM functional feature
 id := counter.toString();  // identifier of functional feature
 description := self.name;  // name of functional feature
 // actor of functional feature
 entity := actors->selectOne(actor | actor.description = aName);
 action := fullOperation;  // action of functional feature
 _object := self._class.name;  // object of functional feature
 executorIsSystem := true;  // mark - if actor is executor of functional feature
 // result of functional feature
 IF (rName <> null) THEN { result._result := rName; } ENDIF;
 // precondition of functional feature
 IF (cond <> null) THEN { precond := cond; } ENDIF;}
```



id	description	action	result	object	entity	ex
1	Board	Board():Board	Board	Board	//@actors.0	true
2	changeTurn	changeTurn()		Reversi	//@actors.0	true
3	mousePressed	mousePressed(java.awt.event.MouseEvent)		Board	//@actors.1	true
4	ReversiPanel	ReversiPanel():ReversiPanel	ReversiPanel	ReversiPanel	//@actors.0	true
5	abNegascoutDecision	abNegascoutDecision(char,char):MoveCoord	MoveCoord	NegaScoutAgent	//@actors.0	true
6	getWhiteScore	getWhiteScore():int	int	Reversi	//@actors.0	true
7	addPanel	addPanel()		ReversiApplet	//@actors.0	true
8	setGameState	setGameState(int)		Reversi	//@actors.0	true
9	effectMove	effectMove(char,char,int,int):char	char	Reversi	//@actors.0	true
10	compareTo	compareTo(MoveScore):int	int	MoveScore	//@actors.0	true
11	NewGameDialog	NewGameDialog(boolean,java.awt.Frame):NewGameDialog	NewGameDialog	NewGameDialog	//@actors.0	true
12	init	init()		ReversiApplet	//@actors.0	true
13	init	init()		Reversi	//@actors.0	true
14	getIsCompTurn	getIsCompTurn():boolean	boolean	Reversi	//@actors.0	true
15	MoveScore	MoveScore(MoveCoord,int):MoveScore	MoveScore	MoveScore	//@actors.0	true
16	getMoveList	getMoveList():java.util.Vector	java.util.Vector	Reversi	//@actors.0	true
17	drawSuggestedPiece	drawSuggestedPiece(java.awt.Graphics,int,int)		Board	//@actors.0	true
18	resetBoard	resetBoard()		Reversi	//@actors.0	true
19	mouseClicked	mouseClicked(java.awt.event.MouseEvent)		Board	//@actors.1	true
20	paint	paint(java.awt.Graphics)		Board	//@actors.0	true
21	evaluateBoard	evaluateBoard(char,char,char):int	int	Evaluation	//@actors.0	true
22	isEffectedPiece	isEffectedPiece(int,int):boolean	boolean	Reversi	//@actors.0	true
23	drawEffectedPiece	drawEffectedPiece(java.awt.Graphics,int,int)		Board	//@actors.0	true
24	getNextMove	getNextMove()		Reversi	//@actors.0	true
25	mouseEntered	mouseEntered(java.awt.event.MouseEvent)		Board	//@actors.1	true
26	drawNewPiece	drawNewPiece(java.awt.Graphics,int,int)		Board	//@actors.0	true
27	getBlackScore	getBlackScore():int	int	Reversi	//@actors.0	true
28	movePiece	movePiece(int,int)		Reversi	//@actors.0	true
29	isValidMove	isValidMove(int,char,char,int):boolean	boolean	Reversi	//@actors.0	true
30	findParentFrame	findParentFrame():java.awt.Frame	java.awt.Frame	ReversiApplet	//@actors.0	true
31	stateChange	stateChange()		Reversi	//@actors.0	true
32	drawPiece	drawPiece(int,java.awt.image.BufferedImage,java.awt.Graphics,int)		Board	//@actors.0	true
33	resetEffectedPieces	resetEffectedPieces()		Reversi	//@actors.0	true
34	findValidMove	findValidMove(char,char,boolean):java.util.ArrayList	java.util.ArrayList	Reversi	//@actors.0	true
35	ReversiFrame	ReversiFrame():ReversiFrame	ReversiFrame	ReversiFrame	//@actors.0	true
36	mouseReleased	mouseReleased(java.awt.event.MouseEvent)		Board	//@actors.1	true
37	initComponents	initComponents()		ReversiPanel	//@actors.0	true
38	setIsCompTurn	setIsCompTurn(boolean)		Reversi	//@actors.0	true
39	findMove	findMove(char,char):MoveCoord	MoveCoord	NegaScoutAgent	//@actors.0	true
40	setEffectedPiece	setEffectedPiece(int,int)		Reversi	//@actors.0	true
41	getInstance	getInstance():Reversi	Reversi	Reversi	//@actors.0	true
42	encode	encode(int,int):String	String	MoveCoord	//@actors.0	true
43	calScore	calScore()		Reversi	//@actors.0	true
44	initComponents	initComponents()		ReversiApplet	//@actors.0	true
45	addToMoveList	addToMoveList(int,char,int)		Reversi	//@actors.0	true
46	mnuNewGameActionPerformed	mnuNewGameActionPerformed(java.awt.event.ActionEvent)		ReversiFrame	//@actors.1	true
47	btnStartGameActionPerformed	btnStartGameActionPerformed(java.awt.event.ActionEvent)		NewGameDialog	//@actors.1	true
48	isNewPiece	isNewPiece(int,int):boolean	boolean	Reversi	//@actors.0	true
49	getGameState	getGameState():int	int	Reversi	//@actors.0	true
50	mnuNewGameActionPerformed	mnuNewGameActionPerformed(java.awt.event.ActionEvent)		ReversiApplet	//@actors.1	true
51	init	init()		Board	//@actors.0	true
52	newGame	newGame()		Reversi	//@actors.0	true
53	main	main(String)		ReversiFrame	//@actors.0	true
54	getBoard	getBoard():char	char	Reversi	//@actors.0	true
55	abNegascout	abNegascout(int,char,int,char,int):MoveScore	MoveScore	NegaScoutAgent	//@actors.0	true
56	init	init()		ReversiPanel	//@actors.0	true
57	update	update(Object,java.util.Observable)		ReversiPanel	//@actors.0	true
58	MoveCoord	MoveCoord(int,int):MoveCoord	MoveCoord	MoveCoord	//@actors.0	true
59	genPriorityMoves	genPriorityMoves(char,char):java.util.ArrayList	java.util.ArrayList	Evaluation	//@actors.0	true
60	initComponents	initComponents()		NewGameDialog	//@actors.0	true
61	Reversi	Reversi():Reversi	Reversi	Reversi	//@actors.0	true
62	btnCancelActionPerformed	btnCancelActionPerformed(java.awt.event.ActionEvent)		NewGameDialog	//@actors.1	true
63	setMessage	setMessage(String)		ReversiPanel	//@actors.0	true
64	mouseExited	mouseExited(java.awt.event.MouseEvent)		Board	//@actors.1	true
65	initComponents	initComponents()		ReversiFrame	//@actors.0	true

Fig. 3. The obtained functional features.

Fig. 4 illustrates one of the obtained UML sequence diagrams by using tool “Visual Paradigm for UML”. In Fig. 3 the first message (findMove) is represented as functional feature 5 and the second message (abNegascoutDecision) as functional feature 39. Relationship between them (from 39 to 5) is provided in the Fig. 5.

It is possible to use a TFM editor in the IDM tool, where the TFM functional features can be visualized. In the next figures all functional features from Fig. 3 are represented with supplementations. Relationships between functional features can be manually added, using the logic of the transformation algorithm as it is shown in Fig. 5. Changes and supplements can be added to the TFM. The TFM graph and TFM model are synchronized – if some changes are made in the graph, after that these changes will be implemented in the model. The example of adding process of relationships between functional features step by step is provided in [2].

All the UML sequence diagrams have been overviewed in order to manually obtain cause-and-effect relations. Lifeline objects of primitive types have been ignored and have not been used during manual execution of the transformation algorithm. Lifeline objects from an external library have also been ignored. It is necessary in order to exclude platform specific information from the model.

Fig. 5 displays the TFM graph with manually obtained relationships between functional features. Some of the cycles formed by them in the TFM model are the following:

- 41-52-18-33-41 – a process of creation of a new game is shown in this cycle. The new game can be started when human player chooses to start it. It can be done at any moment during the game;
- 41-52-18-33-31-2-24-39-9-41 – the main cycle, a process of the game with assignment of turns between players and making a move is defined in this cycle;
- 24-31-2-24 – a process of assignment turns between players is represented in this cycle;
- 45-31-2-24-39-9-45 – a process of finding the effective turn (that leads to victory) for computer player is shown in this cycle;
- 41-28-9-41 – a process of making the effective turn (that leads to victory) by computer player is represented in this cycle;
- 26-32-23-17-26 – cycle defines the process of drawing a visual representation of board and figures of the game, suggested (possible moves for the human player) as well as selected field (selected move by human player) on the game board.

The obtained TFM cannot be considered as the final version, because it is necessary to define all names in human understandable language in the model. It is necessary to delete those cause-and-effect relations which do not correspond to the necessary MDA view, of course if this does not destroy the logic of software system and does not delete important relationships between functional features.

Fig. 6 illustrates the isolated sets of functional features. The TFM does not have to contain these isolated sets of vertices.

Fig. 7 represents isolated functional features. The first column with functional features is the performance of events that depend on the user. They can be defined as overridden operations with or without the body in the source code and the UML sequence diagram uses these operations for visualization. The second column with functional features is the checking of conditions, which are used as conditions in frames from the UML sequence diagram. This operation returns the variable of type “Boolean” or mark for recognizing some activity. The last column of functional features is isolated, because some of them belong to classes, which are the interfaces (in terms of the Object-Oriented programming (OOP)). The interfaces are not considered in this transformation, because the class needs to contain all realizations of operations of interfaces.

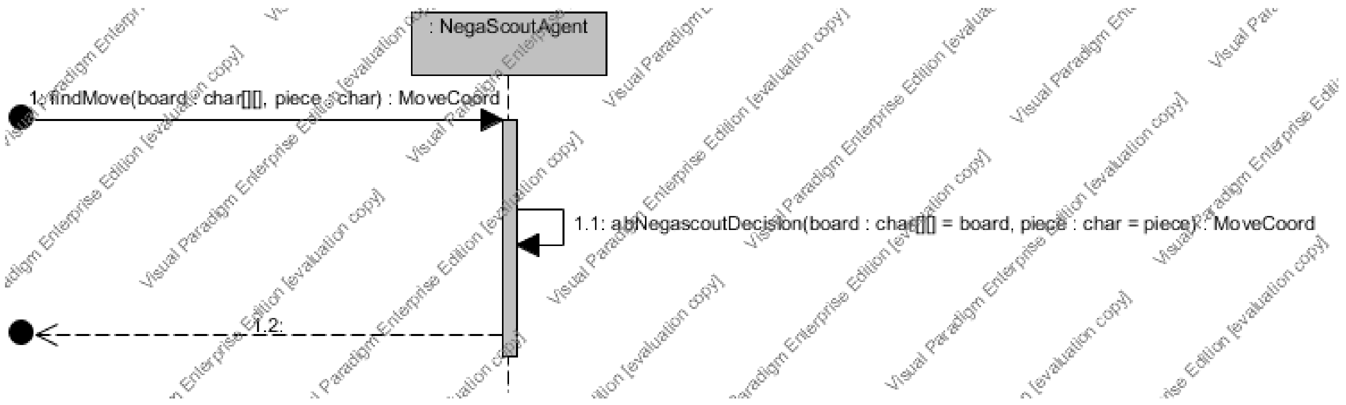

: NegaScoutAgent
1: findMove(board : char[][], piece : char) : MoveCoord
1.1: abNegascoutDecision(board : char[][] = board, piece : char = piece) : MoveCoord
1.2:

Fig. 4. The example of UML sequence diagram.

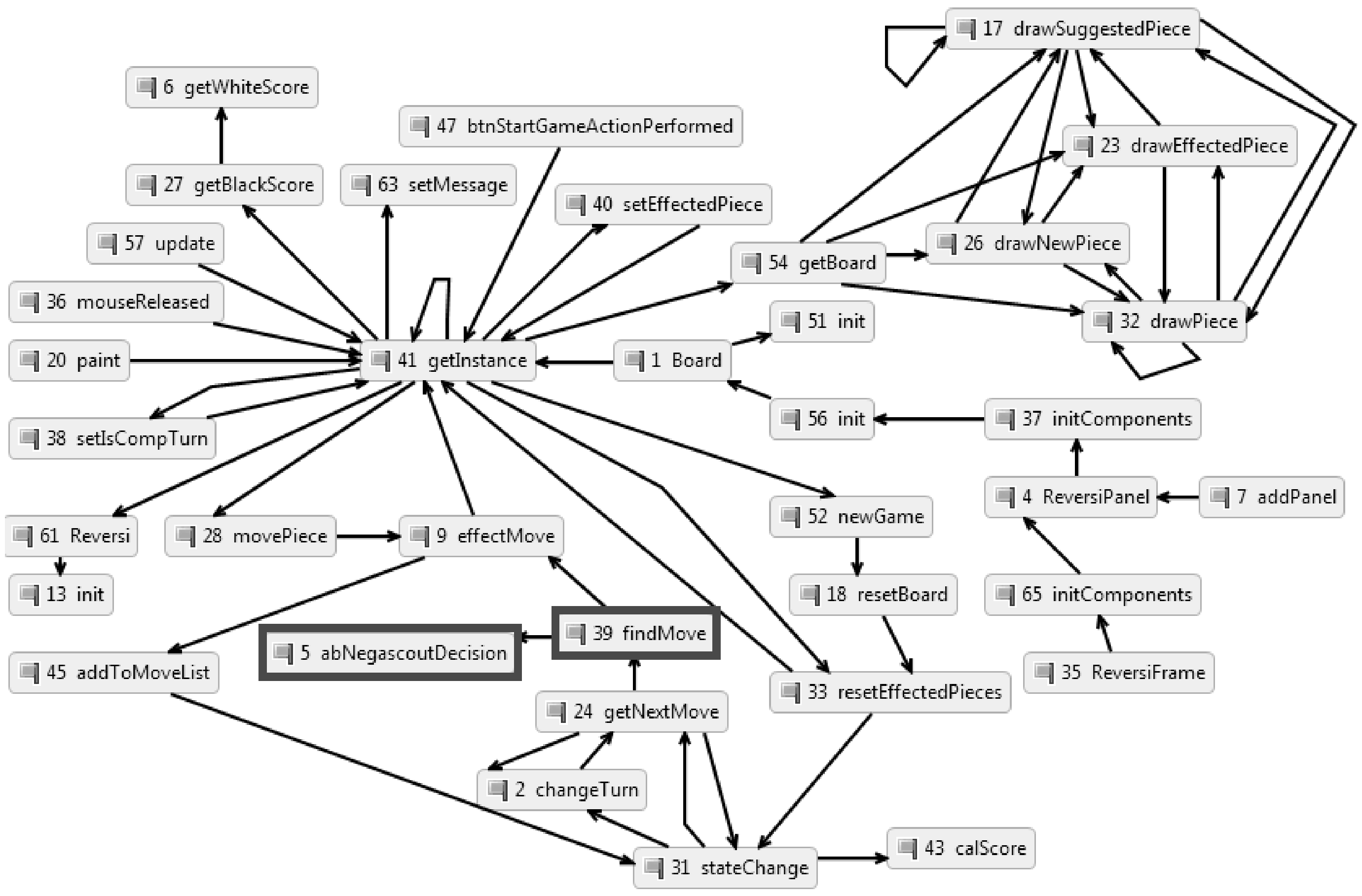


Fig. 5. Manually obtained cause-and-effect relations between functional features.

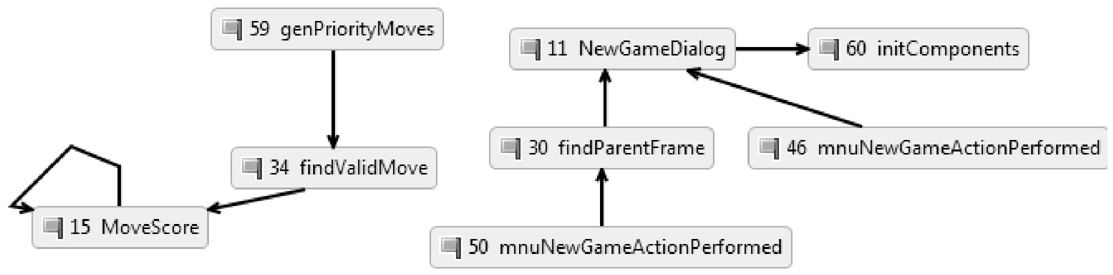


Fig. 6. Isolated set of functional features.

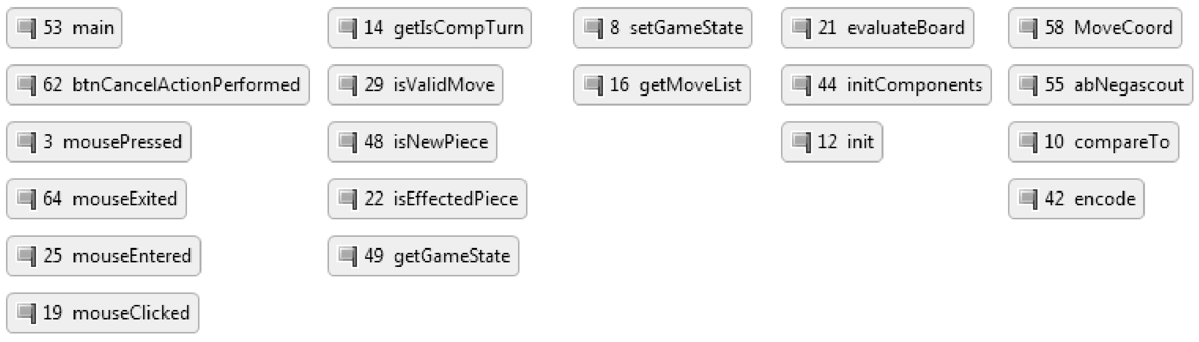


Fig. 7. The isolated functional features.

V. Discussion

The tool “Visual Paradigm for UML” can be used as a standalone tool or as a tool integrated into the Eclipse platform. The usage of the standalone version of the tool provides more functions and the user interface is more understandable. The UML class diagram can be generated directly from the source code, it is not necessary to transfer classes to the main window from the existing source code. Each UML sequence diagram represents an inside action of the operation and the obtainment of one unit of the UML sequence diagram can be received at once (it is impossible to create all UML sequence diagrams at once in this tool). Problems can appear if many classes and operations exist in the source code, but it is good for events when it is necessary to see how to work only some operations.

The information of class and sequence diagrams is provided separately in the exported UML model. Therefore messages from the sequence diagrams not always match the operations from the class diagram and their identifiers also are different. Identifiers of them are related in situations when operation realizes the message and these relations are given in the UML model.

We were unable to access the attribute “xmi:id” which identifies elements of the model during our experiments with the QVTo and by looking for information about it in the documentation. Therefore the identifiers of some necessary elements cannot be known and used. It is a disadvantage, because in case of a message from the UML sequence diagram it is necessary to know the related operation from the UML class diagram for supplementing the TFM functional feature with the information from this operation.

The limitations of the transformation are the following:

- The obtained information may be incomprehensible, because the source code can be written in a bad manner, for example, it could be impossible to understand the operation role only by its name. In this case the generated UML model from the source code will not be comprehensible. Likewise the TFM model will not be readable and full and more time will be necessary for supplementation of the TFM. One of the possible solutions is to refactor parts of the legacy source code or rewrite the whole system – as a result a more readable code should produce better results during the transformation. The development of new software is time-consuming and costly process, because it would be necessary to pay for the entire development process, so it is not always an acceptable solution;
- The system may contain legacy source code (such as operations, classes and other parts) that is not used at all during execution – it would appear in the TFM as isolated vertices;
- The RE tool can acquire incomplete information of the source code. Therefore some information of the software system can be lost.

To summarize, some unclear operations in the legacy software source code are transformed to the UML sequence diagram as messages and additional analysis of source code is necessary for understanding the logic of these operations. Some relationships between functional features are absent, because some operations from the UML class diagrams are used as conditions (in frames) in the UML sequence diagrams. These operations as conditions are not examined in the transformation algorithm. Some operations consist of empty field without any actions in it (e.g. events, mandatory operations from the extended class).

VI. Conclusion

The legacy software system may exist without any documentation. In this case RE could be used for obtaining the visualization of its code in the form of diagrams or models. Of course, it is a risk, because it is not known how precisely the names of the class operations and other elements are defined. It is a problem that has not yet been solved. In the successful case, a domain model can be obtained from this model and could be appropriate for the analysis of the software system.

The TFM backward derivation (transformation) process from the source code is defined in this research. The tool “Visual Paradigm for UML” is chosen and used for generating UML diagrams and exporting UML models. The transformation algorithm to the TFM from the UML model is defined. The part of the transformation algorithm is automated using the QVT operation language. Functional features are obtained automatically from the UML model. After that cause-and-effect relations are manually added, using the IDM tool and following the transformation algorithm. The result is visualized and supplemented using the IDM tool. Positive and negative effects of the TFM backward derivation process are discussed.

The future research direction is related to the improvement and development of the transformation algorithm. The meta-model of the TFM needs to be supplemented with logical operations between the cause-and-effect relations and elements, which are described in the mappings rules to the TFM from the UML sequence diagram. It is necessary that the definitions of names of the functional features are redefined automatically. Parsing of the XMI file with information of the generated UML model to the new structure of the UML model for accessing and adding information needs also be considered in the future.

References

[1] V. Ovchinnikova and E. Asnina, “The Algorithm of Transformation from UML Sequence Diagrams to the Topological Functioning Model,” in *Proceedings of 10th International Conference on Evaluaton of Novel Approaches to Software Engineering*, Barcelona, Spain, 29-30 April, 2015. Portugal: SciTePress, 2015, pp. 377–384. doi: http://dx.doi.org/10.5220/0005476603770384

[2] V. Ovchinnikova, “Research on Backward Derivation of the Topological Functioning Model from Source Code,” M.S. thesis, Riga Technical University, Riga, Latvia, 2015.

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

[4] J. Osis, E. Asnina and A. Grave, “MDA Oriented Computation Independent Modeling of the Problem Domain,” in *Cesar Gonzalez-*

Perez, Leszek A. Maciaszek (eds.) Proceedings of the 2nd International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2007), Barcelona, Spain, July 23–25, 2007. Barcelona: INSTICC Press, 2007, pp. 66–71.

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

[6] E. Asnina and J. Osis, "Computation Independent Models: Bridging Problem and Solution Domains," in *Proc. of the 2nd Int. Workshop on Model-Driven Architecture and Modeling Theory-Driven Development (MDA & MTDD 2010), in conjunction with ENASE 2010*, Athens, Greece, July 22–24, 2010. Lisbon: SciTePress, 2010, pp. 23–32.

[7] L. Favre. (2012, March). *MDA-Based Reverse Engineering* [Online]. Available: http://www.intechopen.com/books/reverse-engineering-recent-advances-and-applications/mda-based-reverse-engineering [Accessed: Oct. 7, 2015].

[8] J. Osis and E. Asnina, "A Business Model to Make Software Development Less Intuitive," in *M. Mohammadian (ed.) Proceedings of the 2008 International Conference on Innovation in Software Engineering [ISE08]*, Vienna, Austria, December 10-12, 2008. Los Alamitos: IEEE Computer Society CPS, 2008, pp. 1240-1245. doi: http://dx.doi.org/10.1109/cimca.2008.52

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

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

[11] U. Donins, "Topological Unified Modeling Language: Development and Application," PhD thesis, Riga Technical University, Riga, Latvia, 2012. Riga: RTU Press, 2012, 224 p.

[12] V. Ovchinnikova and E. Asnina, "Reverse Engineering Tools for Getting a Domain Model within TFM4MDA," in *Proc. of the 11th Int. Baltic Conf. on Databases and Information Systems Baltic DB&IS 2014 "Databases and Information systems"*, Tallinn, Estonia, June 8–11, 2014. Tallinn: Tallinn University of Technology Press, 2014, pp. 417–424.

[13] J. Osis and A. Slihte, "Transforming Textual Use Cases to a Computation Independent Model," in *Model-Driven Architecture and Modeling Theory-Driven Development: Proc. of the 2nd Int. Workshop (MDA & MTDD 2010)*, Athens, Greece, July 22–24, 2010. Lisbon: SciTePress, 2010, pp. 33–42.

[14] A. Slihte, J. Osis and U. Donins, "Knowledge Integration for Domain Modeling," in *Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development (MDA & MDSD 2011)*, Beijing, China, June 8–11, 2011. Lisbon: SciTePress, 2011, pp. 46–56.

[15] A. Slihte, "The Integrated Domain Modeling: an Approach & Toolset for Acquiring a Topological Functioning Model," PhD thesis, Riga Technical University, Riga, Latvia, 2015. Riga: RTU Press, 2015. 224 p.

[16] U. Donins, J. Osis, A. Slihte, E. Asnina and B. Gulbis, "Towards the Refinement of Topological Class Diagram as a Platform Independent Model," in *Proc. of the 3rd Int Workshop on Model-Driven Architecture and Modeling-Driven Software Development (MDA & MDSD 2011)*, Beijing, China, June 8–11, 2011. Lisbon: SciTePress, 2011, pp.79-88.

[17] J. Osis and U. Donins, "Formalization of the UML Class Diagrams," in *Evaluation of Novel Approaches to Software Engineering: 3rd and 4th International Conferences ENASE 2008/2009: Revised Selected Papers*, Milan, Italy, May 9–10, 2009. Berlin: Springer-Verlag, 2010, pp. 180-192. doi: http://dx.doi.org/10.1007/978-3-642-14819-4_13

[18] V. Ovchinnikova, E. Asnina and V. Garcia-Diaz, "Relationships between UML Sequence Diagrams and the Topological Functioning Model for Backward Transformation," *Applied Computer Systems*, vol. 16, 2014, pp. 43-52. doi: http://dx.doi.org/10.1515/acss-2014-0012

[19] Samsoft. (2011, April). *Strategy Guide for Reversi & Reversed Reversi* [Online]. Available: http://www.samsoft.org.uk/reversi/strategy.htm [Accessed: Oct. 7, 2015].

[20] Thuy Gia L. (2015, May). *luugiathuy/ReversiGame* [Online]. Available: GitHub, https://github.com/luugiathuy/ReversiGame [Accessed: Oct. 7, 2015].

[21] OMG. (2015). *Business Process Model and Notation* [Online]. Available: http://www.bpmn.org/ [Accessed: Dec. 12, 2015].

[22] i* wiki. (2011, July). *i* Wiki Home* [Online]. Available: http://istar.rwth-aachen.de/tiki-index.php?page=i%2A+Wiki+Home [Accessed: Dec. 12, 2015].

[23] e3 value. (2015). *The r3 value methodology* [Online]. Available: http://e3value.few.vu.nl/e3family/e3value/ [Accessed: Dec. 12, 2015].

[24] The Open Group. (2015). *ArchiMate* [Online]. Available: http://www.opengroup.org/subjectareas/enterprise/archimate [Accessed: Dec. 12, 2015].

[25] ARIS Community. (2015). *Event-driven process chain* [Online]. Available: http://www.ariscommunity.com/event-driven-process-chain [Accessed: Dec. 12, 2015].

[26] L. Favre, "Formalizing MDA-Based Reverse Engineering Processes," in *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications (SERA '08).* Washington: IEEE Computer Society, 2008, pp. 153–160. doi: http://dx.doi.org/10.1109/SERA.2008.21

[27] S. Rugaber and K. Stirewalt. (2004) *Model-Driven Reverse Engineering* [Online]. Available: http://www.cc.gatech.edu/reverse/repository/model Driven.pdf [Accessed: Dec. 12, 2015].

[28] OMG. (2005). *Architecture-Driven Modernization workshop* [Online]. Available: http://www.omg.org/news/meetings/workshops/adm-2005.htm#tutorials [Accessed: Dec. 12, 2015].

[29] V. Khusidman. (2008). *ADM Transformation* [Online]. Available: http://www.omg.org/adm/ADMTransformartionv4.pdf [Accessed: Dec. 12, 2015].

Viktoria Ovchinnikova received the Bachelor's degree in Automation and Computer Engineering in 2013 and Master's degree in Computer Systems in 2015 from Riga Technical University.

Currently she is the first year doctoral student and a Researcher with the Department of Applied Computer Science of Riga Technical University. She is the author of four conference and one journal paper.

Her current research interests include reverse engineering and model-driven software development.

Address: Department of Applied Computer Science, Riga Technical University, Sētas iela 1, Rīga, LV-1048, Latvia;

E-mail: viktorija.ovcinnikova@rtu.lv