DE GRUYTER OPEN
RIGA TECHNICAL UNIVERSITY
Applied Computer Systems
ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
December 2017, vol. 22, pp. 36–46
doi: 10.1515/acss-2017-0016
https://www.degruyter.com/view/j/acss

Workflow Generation from the Two-Hemisphere Model

Konstantīns Gusarovs[1], Oksana Ņikiforova[2]
[1, 2] *Riga Technical University, Latvia*

***Abstract* – Model-Driven Software Development (MDSD) is a trend in Software Development that focuses on code generation from various kinds of models. To perform such a task, it is necessary to develop an algorithm that performs source model transformation into the target model, which ideally is an actual software code written in some kind of a programming language. However, at present a lot of methods focus on Unified Modelling Language (UML) diagram generation. The present paper describes a result of authors' research on Two-Hemisphere Model (2HM) processing for easier code generation.**

***Keywords* – Code generation, model transformation, system modelling, two-hemisphere model.**

I. Introduction

Business process modelling is one of the most popular trends in software development [1], and distinct kinds of models are used at various stages of software development process [2]. However, it is crucial to mention that most often models are applied to the initial business analysis process and are not reused at other stages of software development.

System model is usually a set of diagrams with specific notations defined for each specifying its syntax and semantics. Various notations exist now and researchers are currently developing new notations as well as working on transformations to make existing ones usable in practice [1]. One of the oldest diagram notations is Data Flow Diagram (DFD) notation developed in 1974 [3]. It describes both a process execution sequence and data flows inside the system, i.e., which process requires which data and what is produced as a result of process invocation. Two-Hemisphere Model [4] first presented in 2004 is like the DFD with several improvements – in addition to the information DFD contains, it also focuses on the presentation of the data structures used in the target software system and gives an ability to assign potential objects-performers of their operations, which is vital for successful implementation.

Since the Two-Hemisphere Model is based on a well-known (as well as well-studied and often used) model, it is logical to focus on its transformation in favour of other models. Moreover, as it was mentioned before, in an ideal case a transformation target is a source code of a computer program. To define a transformation from the source model to the target one, it is necessary to define what both models are. For the source model (2HM) a definition is provided in form of notation and rules that are described later in the paper. To define the target model, the authors would like to cite ISO/IEC 2382:2015 standard [5]. The first definition from it is a "computer program", which should be a result of every software development process. ISO/IEC 2382:2015 defines it as a "syntactic unit that conforms to the rules of a particular programming language and that is composed of declarations and statements or instructions needed to solve a certain function, task, or problem". By analysing this definition, it is possible to define the two main parts of the target model:

- declarations;
- statements or instructions needed to solve a certain function, task, or problem.

Declarations are used to describe data structures and variables used in the target model. Several studies in 2HM transformation area covered in [6]–[8] propose several approaches to obtain this part of a target model. Statements or instructions, however, are not currently covered by existing methods. By further analysing the ISO/IEC 2382:2015, it is possible to see that those can also be described as an algorithm – "a finite ordered set of well-defined rules for the solution of a problem". By combining these definitions, it is possible to define the second part of a target model as "a finite ordered set of statements or instructions" or "a sequence of statements or instructions". Thus, it is possible to say that the transformation algorithm should generate:

- data structure and variable definitions – the static aspect;
- sequence of instructions or statements that use the former part to solve a certain problem – the dynamic aspect.

While the first part of the result has already been examined in several studies, the second one is not being covered at this moment. The paper presents the results of the authors' research on solving this task.

The paper is structured as follows. In the second section, an introduction to the 2HM notation and existing approaches is given. The third section presents the approach proposed by the authors in this paper. Section 4 provides proof of the approach being able to preserve the original information presented in the model. In the fifth section, an example of the proposed technique application to the 2HM model is provided. Finally, the sixth section contains the authors' conclusions as well as covers several areas of the future research.

II. Background of the Two-Hemisphere Model Driven Approach and Related Research

As it was already mentioned, the Two-Hemisphere Model was first presented by Oksana Nikiforova and Marite Kirikova in 2004 [4]. Later, various approaches to transform the Two-Hemisphere Model into the UML class diagram were developed [6]–[8]. UML class diagram [9] allows representing the static part of the system by defining the classes and relations between them. Now, there are a lot of methods to generate a code from the UML class diagrams – since this is a straightforward transformation that consists of a lot of one-to-one transformation rules that describe how the class diagram entities are converted to the code entities.

In some aspects, it is possible to relate the Two-Hemisphere Model to Business Process Model and Notation (BPMN) [10]. Both models are built around business process description and focus on what is happening in the system. However, from the authors' point of view, the 2HM model is easier to work with since it consists of fewer elements and its notation does not dictate strict rules on the layout of the diagram, which makes it more readable. Simple notation also means that the model processing algorithms are easier to define and improve. In future, it would also be possible to reuse these techniques for other kinds of process/data flow diagrams, since they follow the same rules in the definition. Another question one might ask is, why not generate the code from the UML sequence diagram [9], which also describes the dynamics of the system. The authors would like again to point out to the fact that the UML sequence diagram has a more complex notation and can be transformed into the code and vice versa. In some aspects, it is possible to say that the UML sequence diagram might be equal to the code, since it contains all the necessary information, which, in turn, means that its creation requires greater analytical skills. The 2HM model can be read and understood not only by system analysts, but also by business representatives, which makes it a more favourable source model than BPMN and UML diagrams.

There are also several studies on the dynamic aspect of the target model [11], [12]. Now, these studies focus on the UML sequence diagram [9] generation from the Two-Hemisphere Model. The UML sequence diagram is used to present the dynamic aspect of the system; however, its transformation to the code is not as well researched as UML class diagram transformation is. Thus, while it is possible to obtain the sequence diagram of a certain quality, its transformation to the actual source code might be a challenging task. This, in turn, raises the necessity to define an algorithm that would allow obtaining the source code directly or through a more suitable intermediate model. Such an attempt was performed by the authors of paper [13] by reusing some of the ideas from [12]. However, several problems still exist in the proposed algorithm – those will be described later in the present paper.

To understand the problems of the above-mentioned transformation as well as the proposition that authors make in this paper, it is necessary to understand the notation of the Two-Hemisphere Model. The notation itself is presented in Fig. 1 and described in the next paragraphs.

The Two-Hemisphere Model is a concatenation of two different diagrams – a business process diagram, which is like the above-mentioned data flow diagram, and a concept diagram.

Business process diagram in Fig. 1 is marked as G_1 and its notation uses the ideas of DFD [3]. It consists of two types of elements – processes and data flows. Processes show what happens inside the system, while data flows serve for several purposes. They are used to interconnect the processes showing which process accepts which data and which data, in turn, it produces. Data flows also define the sequence, in which processes are executed, thus displaying the dynamic capabilities of the modelled system.

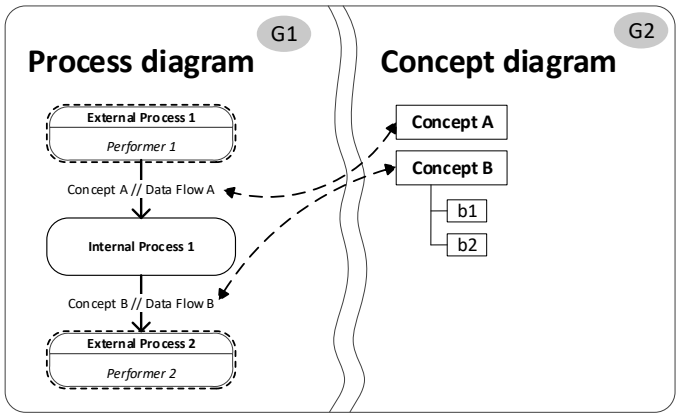


Fig. 1. Notation of the Two-Hemisphere Model.

Concept diagram (G_2 in Fig. 1) is a set of concepts that represent different data types present inside the system being modelled. Each concept might have 0-n attributes and each attribute might represent a primitive data type (such as an integer, floating point number or character sequence), other concept or collection of the above-mentioned types. An important aspect to be noted here is the fact that a concept diagram does not define any kinds of relationships between its elements.

Both parts of the Two-Hemisphere Model are interconnected. This idea presented in [4] is what distinguishes the 2HM approach from the simple use of a business process diagram and conceptual model of any kind. Interconnection is established via assigning concepts to the data flows – each data flow might carry one or more concepts, thus representing which information business process consumes and which information is produced as a result of process invocation.

It is logical to assume that concepts can be transformed into the classes of UML class diagram [9] and, as it has already been mentioned, there are several studies in this area defining algorithms for such a transformation. These algorithms also offer approaches for defining relationships between target classes [6]–[8]. Since the UML class diagram can be used to generate an actual source code and multiple tools to ensure that this way of code generation exist (one of the examples could be SPARX Enterprise Architect CASE tool [14] that allows for source code generation in multiple programming languages), it is safe to say that the 2HM approach covers the aspect of source code generation for static system features.

However, in case of code generation for the actual algorithm (or use case) it is also necessary to define what and in which order should happen. This aspect of code generation in the 2HM approach is not studied very well at the moment. As it has already been mentioned, several approaches have been defined [11], [12] and an attempt to generate the simplified LISP source code has already been performed [13], but these approaches suffer from multiple problems – such as missing or, otherwise, excess information being produced during the transformation. This is due to the approaches that were used during these studies – authors tried to use the existing mathematical methods without considering specifics of the 2HM model. However, these studies were vital for the development of the approach presented in this paper. The next section describes this approach in detail.

III. Workflow Generation from the Two-Hemisphere Model

While describing the approach, the authors would like to point out the fact that it is not necessary to consider a concept diagram at all – it serves other tasks and is not necessary to define the sequence of instructions and statements that are the part of the resulting model, i.e., the source code. In this case, only a business process model is important, since it contains all the necessary information – a sequence in which processes are invoked and data these processes consume and produce. While these data are presented in form of data flows, it is not important that these data flows are holding the information of the concepts – it is possible to use the information in future when a sequence of statements or instructions is defined.

By analysing the structure of the business process diagram, it is possible to note that this diagram is nothing else but a directed multigraph with cycles [15], so graph processing algorithms might be used for the transformation. It is also possible to note that the idea of business process modelling itself is to represent different states in which the system might reside and data (and possibly conditions) that are necessary for the system to enter such states. This point of view gives the idea that the business process diagram could also be considered a Finite-State Machine (FSM) [16], which is also the directed multigraphs allowing cycles. This idea was used in [12] and [13], and in conjunction with a transitive closure [15] method it allowed converting the business process diagram into the regular expression.

Important note about regular expression is the fact that it is a linear structure, i.e., a sequence of tokens, which corresponds to the expected result – a sequence of statements and instructions. However, while the result of transitive closure method application to the business process model is correct, it contains excess information, which is noted in [13]. Source code generated from such an expression would suffer from the duplications. For example, a body of the loop would be generated twice – before the actual loop and inside it. Such a source code would be hard to debug and maintain. Thus, pure mathematical methods are not enough for business process diagram transformation to the source code.

However, the idea of graph processing algorithm utilisation during the transformation is valid. It is only necessary to define or develop correct algorithms that would keep all the source information intact and be able to transform the directed multigraph (or FSM) to the linear sequence of statements or instructions.

To successfully complete such a task, the authors propose using a process-centric approach, i.e., using business processes from the business process diagram as the main building block for the linearisation task purpose. Data flows in this case determine process inputs, outputs and execution sequence. To achieve this purpose, it is necessary to perform two operations:

- For each process in the business process model its inputs and outputs are taken into account and the model is transformed into the vertex in a graph forming something like the function signature. Because of this transformation, data flows are removed from the diagram becoming parts of the processes and edges in the graph that do not carry any information. This step of transformation is shown in Fig. 2. Figure 2 represents part of the business process diagram with a single process `P` that has two incoming data flows – `IDF1` and `IDF2` and two outgoing – `ODF1` and `ODF2`. While the structure of the model is kept intact, its edges are now serving only one purpose – process execution order definition. Information about incoming/outgoing data is now part of the process itself.
- After moving data flows inside processes, it is possible to exchange graph vertices and edges. During this transformation, it is possible for new edges to appear in the graph – this happens when the process has several inputs and is totally valid for further transformation steps. Example of such a transformation is shown in Fig. 3. Figure 3 demonstrates the model that has 5 processes, one of which – `P4` – consumes 3 different data flows and produces a single output. Therefore, the transformation graph contains 6 vertices (instead of initial 5) and some duplicated edges.

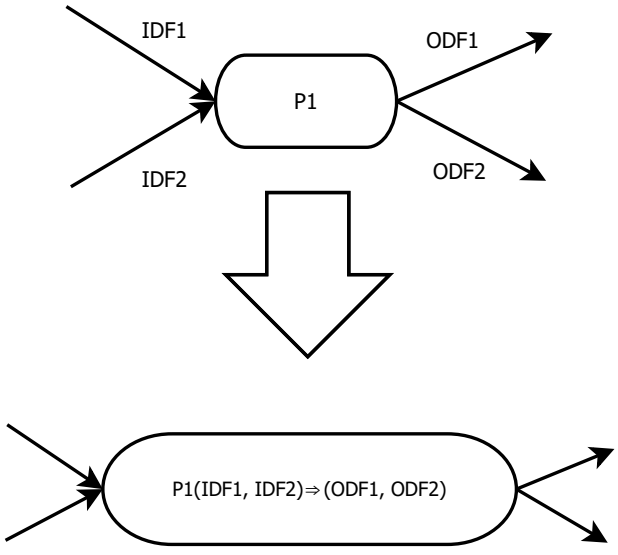


Fig. 2. Process signature definition.

After those two steps performed, the business process diagram turns out to be almost like the FSM, where vertices represent the states system can stay in, but edges show what sequence of actions must be executed for the system to leave one state and enter another. Now it is possible to perform the linearisation task to reduce this FSM into a smaller one. In an ideal case, which is the target state, the graph/FSM should have only two vertices – the initial state and the final state, and a single edge connecting those. To achieve such a situation, the authors have developed 9 different minimisation techniques. Each of these techniques targets a single area in the graph and transforms it. Minimisation/linearisation task, in turn, becomes an iterative process when on each iteration these techniques are being tried against the current state of the graph. If at least one of them succeeds, iteration starts over. During their experiments, which comprised the application of the proposed algorithms to ~100 different process models including various possible cases, the authors found out that these techniques were enough to achieve the target state of the graph with two vertices and a single edge in all the test cases. However, it is possible that such a graph, which cannot be minimised, exists but was never discovered. In this case, after all the minimisation/linearisation iterations it would be possible to apply the above-mentioned transitive closure [15] method to perform the final reduction of the graph. In the next paragraphs, the authors will define all the techniques used for the minimisation purpose and provide examples.

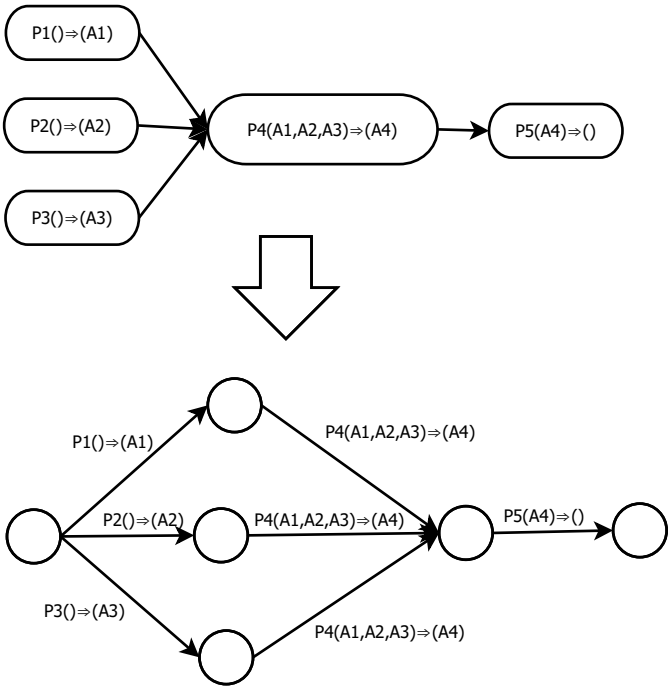


Fig. 3. Vertex-edge exchange process in a graph.

The first and most obvious technique to be used is the elimination of duplicate edges. It is possible to describe it using the following rule:

If there are vertices `A` and `B` in the graph that are interconnected with the same edges `X, Y, … = Z`, then it is possible to eliminate these duplicate edges replacing those with a single one – `Z`. An example of this technique being applied to a graph is provided in Fig. 4. The figure shows a part of the graph being transformed where two vertices are interconnected with the same two edges, and a transformation results in duplicate edges eliminated.

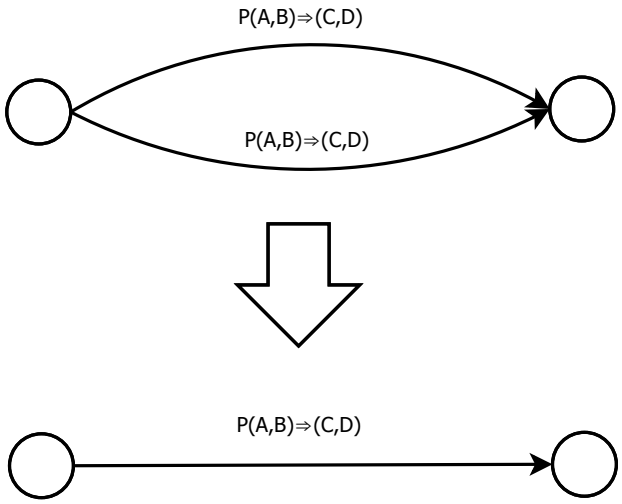


Fig. 4. Duplicate edge elimination.

Next technique also analyses vertices that are interconnected with an edge set. However, in this case all the edges in the set should be different. The authors call this technique "disjunction definition", and it can be described as follows:

If there are vertices `A` and `B` in the graph that are interconnected with the different edges `X, Y, …` such as edges in the set are different and it is not possible to extract a common suffix from these, it is possible to eliminate these edges and replace them with a single edge `(X | Y | …)`. There is a definition in the rule that requires additional explanation. The authors consider that two edges have a common suffix if an action sequence associated with each of the edges ends with the same sequence of actions. The simplest example in this case is duplicate edges mentioned before – it is possible to consider that duplicate edges have a common suffix which is the whole edge. Another example that could be mentioned here is the set of two edges – `(X,Y,Z)` and `(W,Y,Z)`. Edges in this example would have a common suffix – `(Y,Z)`. Disjunction definition technique will not affect the edges that share a common suffix since there are several techniques that are meant specifically for such cases. Example of a disjunction definition technique is provided in Fig. 5. In this case, there is again a graph with two vertices that are interconnected with two edges. In contrary to the previous example, these edges are different; however, they are still eliminated and replaced with a single edge.

To process and transform common suffixes in an edge set, the authors propose two different techniques that are described below. The first technique is used in case when two vertices `A` and `B` exist, and these vertices are interconnected with an edge set sharing a common suffix. In this case, the authors propose introducing new vertex `C` and interconnecting it with `A` and `B` in such a way that `C` is connected with `B` via a single

edge containing a common suffix, while A is interconnected with C via multiple edges that contain different sequences that were left after common suffix extraction. Example of this technique is shown in Fig. 6.

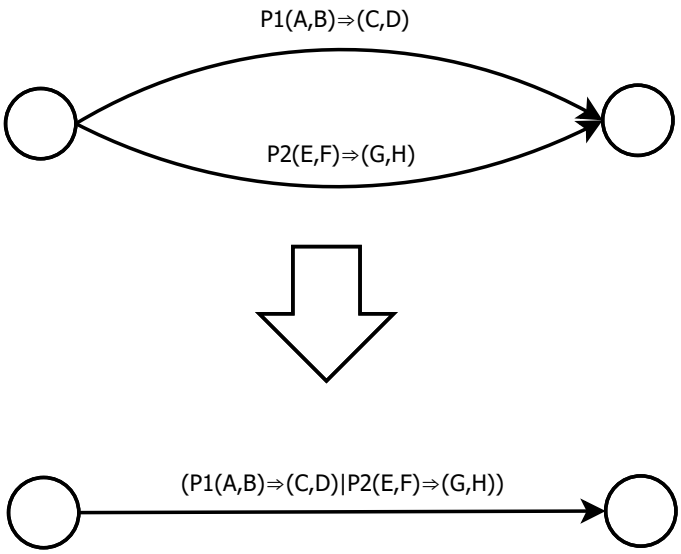


Fig. 5. Disjunction definition.

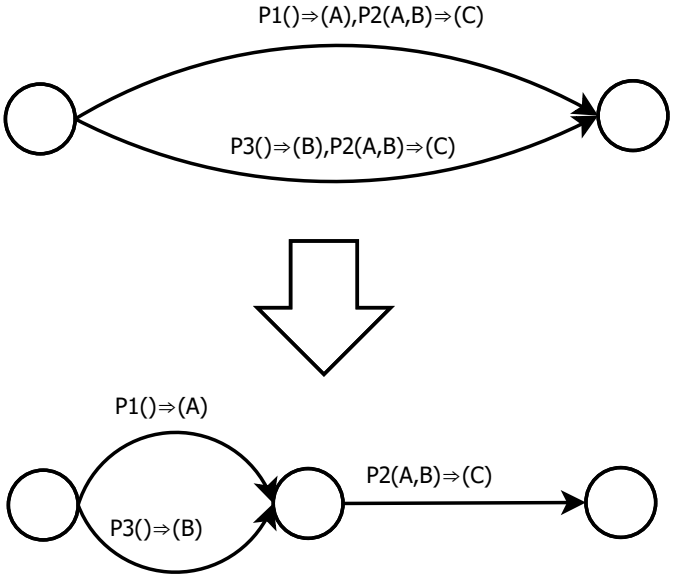


Fig. 6. The first common suffix processing technique.

The second technique used for common suffix processing is different. It analyses only a single vertex A in a graph and an edge set that is incoming inside. If this edge set shares a common suffix, it is possible to create new vertex B that is interconnected with A using an edge carrying a new suffix. Other vertices connected with A before the application of this technique, in turn, will become connected to vertex B via edges carrying parts of original action sequences minus a common suffix part. An example of this technique is shown in Fig. 7.

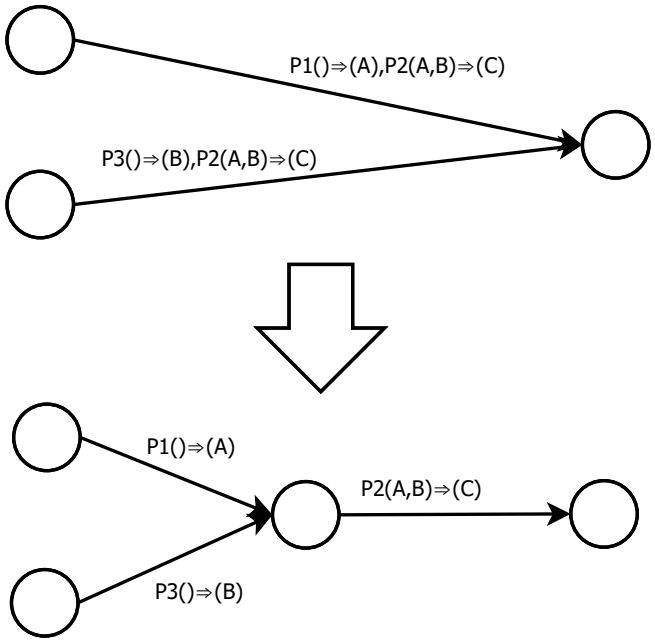


Fig. 7. The second common suffix processing technique.

At first, it might seem that both common suffix processing techniques increase graph complexity by adding new vertices and edges, while the main task of all techniques that the authors define is opposite – to minimise the graph and convert it to a linear structure. However, after the application of these techniques, a graph becomes available for processing with other techniques that will eliminate new additions and, as a result, reduce the complexity of it. For example, in case of the first common suffix processing technique shown in Fig. 6, it is possible to see that part of the graph is now available for application of disjunction definition, while without common suffix processing it would be impossible.

Next two techniques defined by the authors of this paper target cycles and circles that might appear in the graph of the business process model. The first technique searches the graph for vertices A and B that are interconnected with edges X and Y in such a way that a cycle is formed in the graph. If it is possible to find such vertices, then it is possible to merge them into a single vertex forming a circle. In this case, it is necessary to determine which vertex in original process graph was the beginning of the cycle, and which was its ending. This is achieved by analysing, how far vertices are from the initial state of the graph shown in Fig. 3. Since graph transformation to the FSM will always create a single initial and single final state, this task becomes trivial and can be performed by calculating the shortest paths from the initial state to vertices A and B. If one of the paths includes one of these vertices, then the vertex becomes the end of the cycle. Newly created circle, in turn, might be described as (X,Y)* or (Y,X)* depending on the result of this calculation, where * defines the fact that the given sequence of action might be executed zero or more times. An example of this technique is shown in Fig. 8.

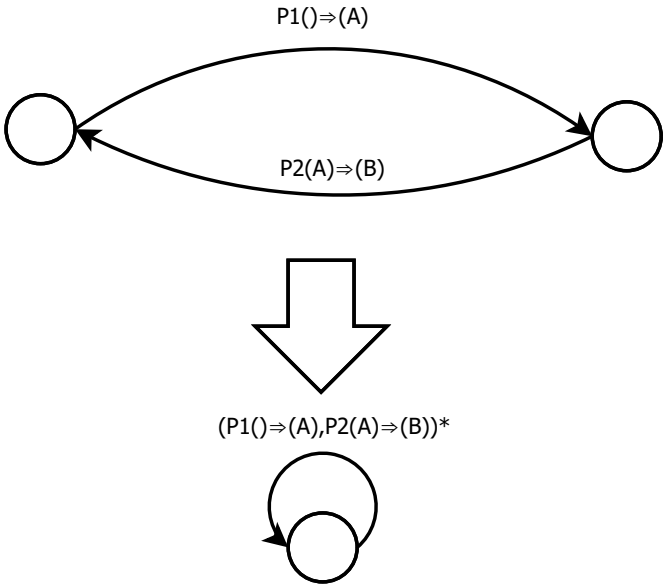


Fig. 8. Cycle elimination by vertex merging.

As a result of this technique application, a circle in the graph is created. The next technique, in turn, targets circles in the graph to replace. This technique searches for vertex `A` in a graph, such that it has only one incoming edge `X`, only one outgoing edge `Y` and circle `Z`. If such a vertex is found, it is possible to remove it from the graph and replace it with an edge `(X,Z*,Y)` that will connect `A` predecessor and successor. An example of this technique application is given in Fig. 9.

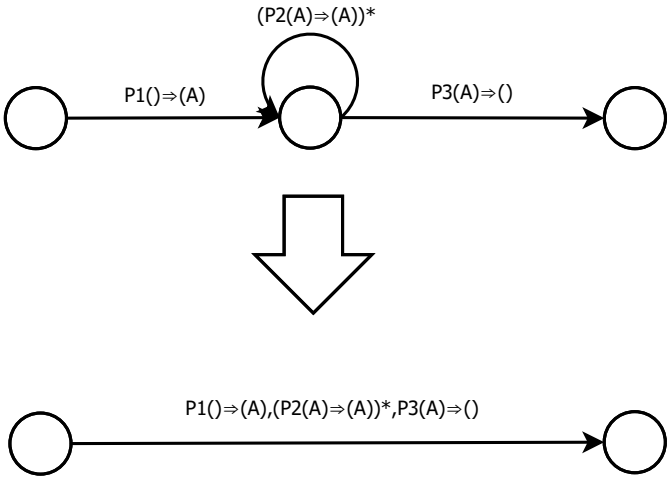


Fig. 9. Circle elimination.

Next, the technique defined by the authors searches the graph for vertices that have only one incoming and only one outgoing edge. These vertices can be removed from the graph and replaced with a new edge that will combine input and output of a removed vertex as a new action sequence. The proposed technique is simple and straightforward, and the example of its application is given in Fig. 10.

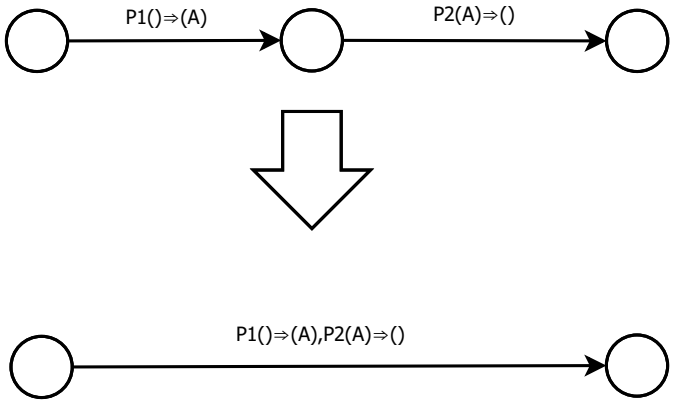


Fig. 10. Vertex elimination.

Next, the graph processing technique searches the graph for vertex `A`, such as it has multiple duplicate outgoing edges `X, Y, … = Z`. If such a vertex is found inside the graph, it is possible to create new vertex `B`, connect `A` to `B` via new edge `Z`, and connect `B` to the original successors of `A`. Again, this technique might seem like one that complicates graph structure instead of simplifying it; however, in this case a degree of outgoing edges for A is lowered, thus serving the minimisation task. An example of this technique is provided in Fig. 11. In this figure, action sequences for the removed vertex grandchildren are omitted since they do not participate in the transformation and do not carry any information that is vital for technique application.

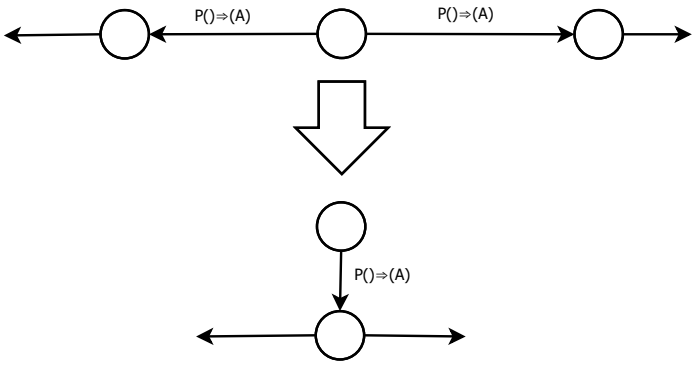


Fig. 11. Duplicate outgoing edge elimination.

During the task of graph minimisation, it is possible to achieve a situation, such that there is edge `X` in the graph that carries no information. One of the examples, when such an edge can appear, is common suffix elimination, e.g., if there were edges `(X,Y)` and `(Y)` that were sharing common suffix `(Y)`, after suffix elimination these edges would be transformed into `(X)`, `()` and `(Y)`. It is possible to note that an empty edge has appeared. Such edges can be safely removed from the graph; however, it is important to preserve information about an original graph structure when performing such an operation. The last technique that the authors would like to describe in this paper targets such situations and removes empty edges. Example of its application is given in Fig. 12. This example also shows that after application of this technique it is possible that a circle will be created in a graph.

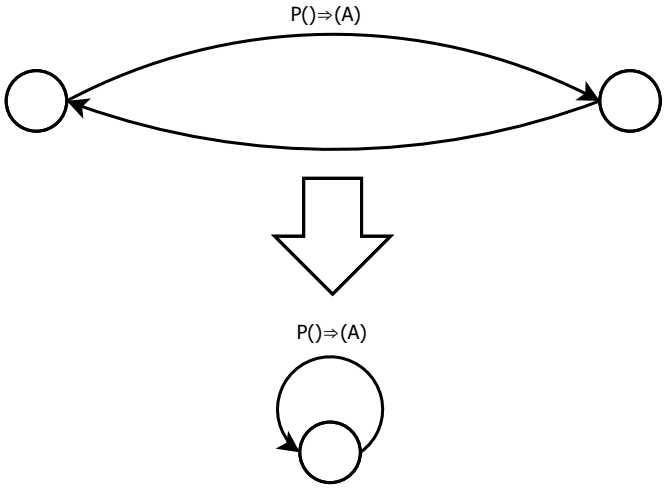


Fig. 12. Empty edge elimination.

After successful application of graph minimisation techniques to the business process diagram, a linear structure is created. This structure resembles the above-mentioned regular expression and can be further used for code generation. The authors call this structure a "workflow model" since it defines what and in which order happens inside the system when performing some task.

Workflow model is very close to the source code in terms of its syntax. For example, the workflow model for the business process diagram part presented in Fig. 9 might look like (see Fig. 13):

```
A = P1()
repeat: {
  A = P2(A)
}
P3(A)
```

Fig. 13. An example of workflow model.

It is possible to see that the workflow model is a starting point for code generation – it has information on variable names, how they are used, what processes and in which sequence are invoked. Loops and potential conditional operators are also detected. However, in the example above it is possible to note that a loop still misses an important piece of information – a condition for looping. The same would apply to the disjunctions – it is not possible to determine if disjointed sequences of actions are executed in parallel, exclusively or together as well as it is impossible to determine under which condition each of the action sequences should be executed. This is a target for the future research, and it seems that it would be necessary to make changes to the 2HM model notation, since its current state makes it impossible to extract such information from the business process diagram for the further transformation purposes. It should be possible to define a condition in a free-text form for each of the processes. Such a condition should determine when the given process is executed. Since the process might be part of the process sequence, this would possibly allow defining finite blocks of statements that are being invoked under a given condition.

Another thing to be considered in the present research is the preservation of all the available information, source model, provided by the 2HM model. It should be prohibited for a transformation method to ignore parts of the initial model, since all the information in the business process diagram is important for the definition of the algorithm it describes. Thus, it is possible to say that a valid transformation method should use and preserve all the original information from the source model in order to make this information part of the target model as well.

When minimising the graph, it is important to preserve the original information it was representing without breaking its structure. This means that graph processing techniques should be "non-breaking" operations that will not lose any information present. In the next section, the authors would like to give proof that techniques they propose meet this condition with one exception that will also be described.

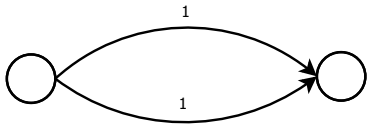
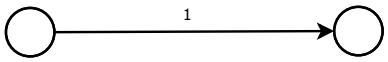
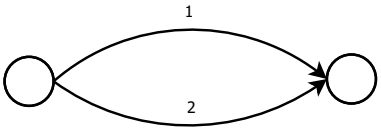
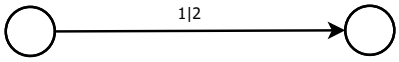
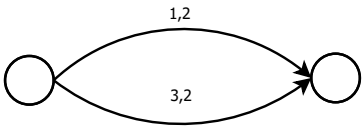
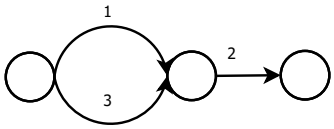
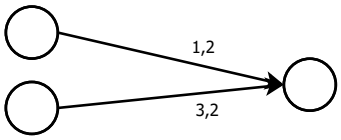
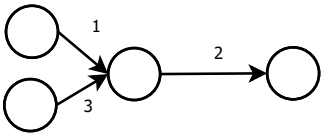
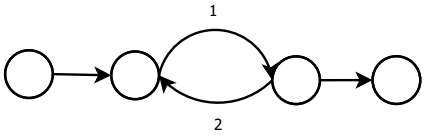
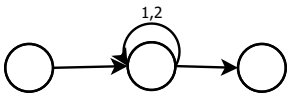
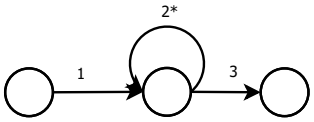
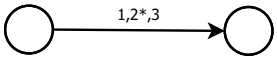
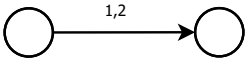
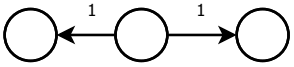
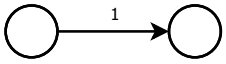
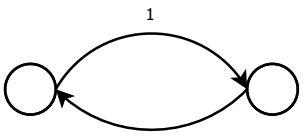
IV. Proof of Information Preservation for the Proposed Minimisation Techniques

To prove that the proposed techniques preserve the original structure of the business process diagram, the authors would like to use the fact that this diagram and FSM that corresponds to it are, in fact, directed graphs. This means that it is possible to define paths in such graphs and analyse that all the paths that were possible before the application of a single technique would still be possible after its application. If it is so, then it is possible to say that a given technique preserves the initial structure of the graph. It is important to note the fact that only edges are included in the path – since only edges carry the information about process invocation. In this section, the authors will describe the use of this proof for all the proposed techniques.

Table I covers proof for all the techniques described in this paper. Its first column contains a technique name, the second column presents a situation in a graph before the application of the given technique, the third column shows a situation after such an application, the fourth and fifth columns, in turn, show paths before and after application of the technique. In this table, the authors omit the information about actual action sequences on the graph edges and use numbers to mark the appropriate parts of the graph.

TABLE I

PROOF OF GRAPH STRUCTURE PRESERVATION

Technique	Graph Before	Graph After	Paths Before	Paths After
Duplicate Edge Elimination			• 1 • 1	• 1
Disjunction Definition			• 1 • 2	• 1 • 2
Common Suffix Processing I			• 1–2 • 3–2	• 1–2 • 3–2
Common Suffix Processing II			• 1–2 • 3–2	• 1–2 • 3–2
Cycle Elimination*			• 1 • (1–2)*	• (1–2)*
Circle Elimination			• 1–2*–3	• 1–2*–3
Vertex Elimination			• 1,2	• 1,2
Duplicate Outgoing Edge Elimination			• 1 • 1	• 1
Empty Edge Elimination			• 1*	• 1*

It is possible to see from Table 1 that all the proposed techniques preserve the initial structure of the graph after their application with only exception of Cycle Elimination technique. Cycle Elimination removes the information from the graph about one of the possible paths. At the current state, the authors think that this is fine, since cycles in the graph correspond to the loops in the appropriate algorithm described by the business process diagram and source code generated from it. By applying the Cycle Elimination technique, information about the loop is preserved; however, one of the possible control flows – premature loop exit is lost. The authors propose marking such situations in a graph so that in the future when the source code is generated, it would be possible to insert an appropriate programming language statement – one that allows exiting the loop, e.g., `break`. For an example given in Table 1, the technique application result in such a case would be `(1,( break|2))*` instead of `(1,2)*`. However, this is the target for the future work, and it is possible that there are other solutions to this problem.

All other graph processing techniques preserve all the paths that were present in the graph before appropriate technique was applied. Thus, it is possible to prove that techniques the

authors propose are "non-breaking" and can be used to transform the business process diagram to a linear structure that, in turn, can be later used for source code generation from the 2HM model.

In the next section of this paper, the authors would like to provide a simple example of how the described graph transformation techniques can be used to convert the initial business process diagram to a workflow model.

V. A Workflow Generation Example

To demonstrate how the proposed techniques can be applied to the 2HM model, the authors would like to refer to a simple example shown in Fig. 14. In this case, the concept diagram part of the 2HM model is omitted since it plays no role in the transformation process. Only the business process diagram is given. This diagram shows a room booking process at a hotel. It starts with a booking request submission that contains details about room preference. After this request is submitted, there are two options: a room that fits the request might be found or request might be rejected due to some reason, e.g., no room meets the criteria, requested dates are not available etc. In case of request rejection, a user is asked to revise the information and submit a new request. This process might repeat multiple times. User might also cancel the booking. In this case, the business process is ended and the algorithm should stop. If a room is found and the request can be served, a user is asked to provide additional information. Finally, the request will be converted to the booking information and stored in the database. This business process diagram despite its simplicity covers multiple possible cases, such as loops and conditions, and, thus, can be used to show the previously described techniques in action. All elements in this diagram are marked with identifiers – processes are marked P1…P8, dataflows – D1…D8.

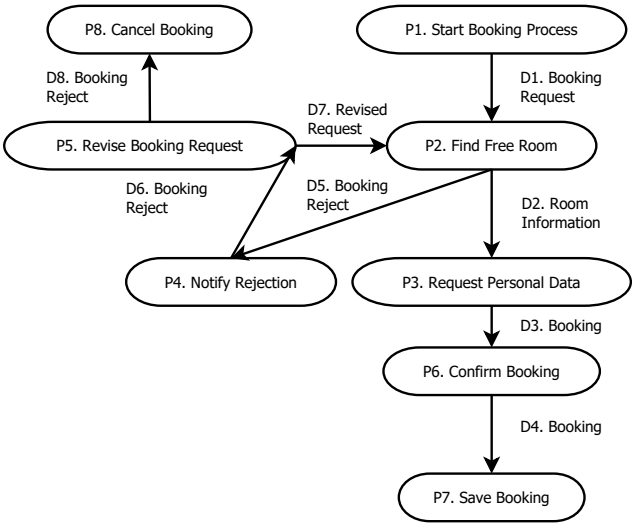


Fig. 14. Sample business process diagram.

As it was described before, to prepare the business process diagram graph for the application of techniques developed by the authors, it is necessary to convert it to the FSM. The appropriate FSM for the sample model is given in Fig. 15.

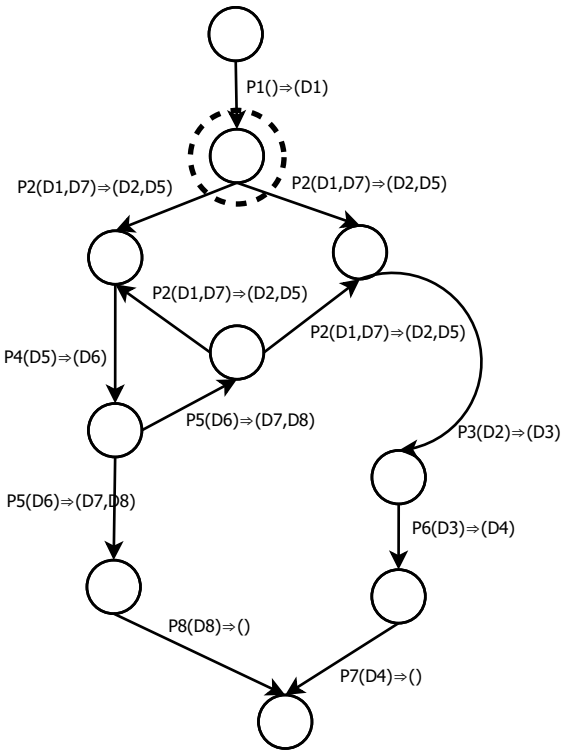


Fig. 15. FSM of sample business process diagram.

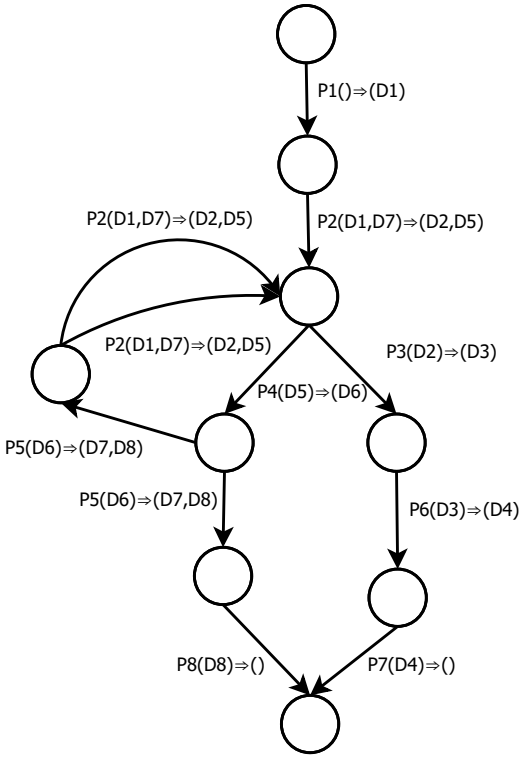


Fig. 16. Application result of Duplicate Outgoing Edge Elimination technique.

This FSM is the directed multigraph that allows for cycles, and it is possible to process it with techniques defined by the authors. Figure 16 shows this graph after Duplicate Outgoing Edge Elimination technique application to a vertex that is marked with a dashed line in Fig. 15.

Next steps will use other techniques to reduce graph complexity and convert it to a linear structure; however, due to limitations of this paper, the authors will omit description of each step and present the results of technique application in form of workflow model that was generated from the initial business process diagram. This model is presented in Fig. 17.

It is possible to see that the resulting workflow model contains both loop and disjunction. However, in this case disjunction means that each of its branches is independent, i.e., only one of these can be executed. In real life cases, there is possible such a situation that both branches of disjunction are executed in parallel. As it has already mentioned, the 2HM model lacks information that would be necessary to identify such cases, and it should be enriched to make the transformation more precise. The same applies to the loops in the resulting model – there is no condition for loop exit.

```
(D1.Booking Request) = P1.Start Booking Process();
repeat: {
  (D2.Room Information, D5.Booking Reject) =
    P2.Find Free Room(D1.Booking Request, D7.Revised Request);
  (D6.Booking Reject) = P4.Notify Rejection(D5.Booking Reject);
  (D7.Revised Request, D8.Booking Reject) =
    P5.Revise Booking Request(D6.Booking Reject);
}
disj{
  case1: {
    P8.Cancel Booking(D8.Booking Reject);
  }
  case2: {
    (D3.Booking) = P3.Request Personal Data(D2.Room Information);
    (D4.Booking) = P6.Confirm Booking(D3.Booking);
    P7.Save Booking(D4.Booking);
  }
}
```

Fig. 17. Generated workflow model.

VI. Conclusion and Future Research

In the paper, the authors have presented a set of techniques that can be applied to the business process diagram, which is part of 2HM model. Using these techniques, it is possible to minimise the initial graph of the business process diagram and convert it to a linear structure that later can be used for source code generation.

As a result, an intermediate model that the authors call the workflow model has been obtained. In the paper, it is represented in a textual form that is like the source code of the program; however, it is a linear structure that consists of action sequences, disjunctions and loops.

The authors provide proof that techniques they offer to use are "non-breaking" and preserve the original structure of the graph during the transformation. However, there is one exception the authors point out in the paper. The exception is Cycle Elimination technique that "loses" the information about possible loop exits. The authors propose an approach to fix this issue; however, it is necessary to perform a detailed analysis of such an improvement to the current state of art. This is one of the targets for the future research.

Another task to be solved is to find out if the proposed minimisation/linearisation techniques can reduce all kinds of business process diagrams to the workflow model. If it is so, these techniques are enough to create an intermediate model suitable for future source code generation. If not, it is necessary to find cases where these techniques do not apply, as well as propose the solutions for such case processing.

As it has already been mentioned, the workflow model can be written in form that is like the source code, which proves the fact that such a transformation is possible. However, it is worth noting that it still misses some vital information such as conditions for loops or disjunctions. Furthermore, it is impossible to distinguish a type of disjunction – its branches are exclusive and only one must be executed, or they are invoked in a parallel and both should be invoked. At the moment, it is impossible to obtain such information due to the fact that the initial model misses it. This leads to another direction of future research – enriching the 2HM model with the necessary information.

As a conclusion, the authors would like to state that the approach described in the paper allows achieving results that are easily convertible to the source code, and further research in this direction should allow for more precise code generation.

REFERENCES

[1] W. M. P. van der Aalst, "Business Process Management: A Comprehensive Survey," *ISRN Software Engineering*, vol. 2013, pp. 1–37, 2013. https://doi.org/10.1155/2013/507984

[2] M. Brambilla, J. Cabot, M. Wimmer, *Model-Driven Software Engineering in Practice*, 1st edition. USA: Morgan & Claypool Publishers, 2012.

[3] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974. https://doi.org/10.1147/sj.132.0115

[4] O. Nikiforova and M. Kirikova, "Two-Hemisphere Model Driven Approach: Engineering Based Software Development," *Lecture Notes in Computer Science*, pp. 219–233, 2004. https://doi.org/10.1007/978-3-540-25975-6_17

[5] ISO/IEC 2382:2015 Information technology – Vocabulary [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en

[6] O. Nikiforova and N. Pavlova, "Development of the Tool for Generation of UML Class Diagram from Two-Hemisphere Model," *2008 The Third International Conference on Software Engineering Advances*, pp. 105–112, Oct. 2008. https://doi.org/10.1109/icsea.2008.37

[7] O. Nikiforova, K. Gusarovs, O. Gorbiks, and N. Pavlova, "BrainTool. A Tool for Generation of the UML Class Diagrams" *Proceedings of the Seventh International Conference on Software Engineering Advances*, Mannaert H. et al. Eds, pp. 60–69, Lisbon, Portugal, November 18–23, 2012.

[8] O. Nikiforova, L. Kozacenko, D. Ungurs, D. Ahilcenoka, A. Bajovs, N. Skindere, K. Gusarovs, and M. Jukss, "BrainTool v2.0 for Software Modeling in UML," *Applied Computer Systems*, vol. 16, no. 1, pp. 33–42, Jan. 2014. https://doi.org/10.1515/acss-2014-0011

[9] Unified Modeling Language (UML) [Online]. Available: http://www.uml.org

[10] BPMN Specification – Business Process Model and Notation [Online]. Available: http://www.bpmn.org

[11] O. Nikiforova, L. Kozacenko, and D. Ahilcenoka, "UML Sequence Diagram: Transformation from the Two-Hemisphere Model and Layout," Applied Computer Systems, vol. 14, no. 1, pp. 31–41, Jan. 2013. https://doi.org/10.2478/acss-2013-0004

[12] O. Nikiforova, K. Gusarovs, and A. Ressin. "An Approach to Generation of the UML Sequence Diagram from the Two- Hemisphere Model," *Proceedings of The Eleventh International Conference on Software Engineering Advances (ICSEA)*, 2016.

[13] K. Gusarovs, O. Nikiforova, and A. Giurca, "Simplified Lisp Code Generation from the Two-hemisphere Model," Procedia Computer Science, vol. 104, pp. 329–337, 2017. https://doi.org/10.1016/j.procs.2017.01.142

[14] UML tools for software development and modelling – Enterprise Architect UML modeling tool [Online]. Available: http://www.sparxsystems.com

[15] T. Koshy, *Discrete Mathematics with Applications*. Academic Press, p. 1042 p., 2003.

[16] "Finite State Machines," 2005 [Online]. Available: http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf

Konstantīns Gusarovs received the Master degree in Computer Systems from Riga Technical University, Latvia, in 2012. At present, he is the third year Doctoral student and Researcher at the Department of Applied Computer Science, Riga Technical University, as well as Java Developer in C.T.Co Ltd. His current research interests include object-oriented software development and automatic obtaining of program code.
E-mail: konstantins.gusarovs@gmail.com

Oksana Ņikiforova received the Doctoral degree in Information Technologies (system analysis, modelling and design) from Riga Technical University, Latvia, in 2001.
At present, she is a Professor at the Department of Applied Computer Science, Riga Technical University, where she has been working since 1997. Her current research interests include object-oriented system analysis, design and modelling, especially the issues in Model Driven Software Development.
E-mail: oksana.nikiforova@rtu.lv