RIGA TECHNICAL UNIVERSITY

Faculty of Computer Science and Information Technology
Institute of Applied Computer Systems

Konstantīns Gusarovs

Doctoral Student of the Study Programme "Computer Systems"

A DEVELOPMENT OF METHOD FOR CODE GENERATION FROM TWO-HEMISPHERE MODEL

Summary of the Doctoral Thesis

Scientific supervisor
Professor Dr. sc. ing.
OKSANA ŅIKIFOROVA

RTU Press
Riga 2021

Gusarovs, K. A Development of Method for Code Generation From Two-Hemisphere Model. Summary of the Doctoral Thesis. Riga: RTU Press, 2021. 40 p.

Published in accordance with the decision of the Promotion Council "P-07" of 4 November 2020, Minutes No. 20-7.

DOCTORAL THESIS PROPOSED TO RIGA TECHNICAL UNIVERSITY FOR THE PROMOTION TO THE SCIENTIFIC DEGREE OF DOCTOR OF SCIENCE

To be granted the scientific degree of Doctor of Science (Ph. D.), the present Doctoral Thesis has been submitted for the defence at the open meeting of RTU Promotion Council on March 1 2021 at 14:30 at the following link: https://rtucloud1.zoom.us/j/91849258957.

OFFICIAL REVIEWERS

Professor Dr. habil. sc. ing. Jānis Grundspeņķis
Riga Technical University, Latvia

Professor Dr. sc. ing. Artis Teilāns
Rezekne Academy of Technologies, Latvia

Teaching Assitant PhD Dušan Savić,
University of Belgrade, Serbia

DECLARATION OF ACADEMIC INTEGRITY

I hereby declare that the Doctoral Thesis submitted for the review to Riga Technical University for the promotion to the scientific degree of Doctor of Science (Ph. D.) is my own. I confirm that this Doctoral Thesis had not been submitted to any other university for the promotion to a scientific degree.

Konstantīns Gusarovs ……………………………. (signature)

Date: ………………………

The Doctoral Thesis has been written in Latvian. It consists of an Introduction; 7 chapters; Conclusion; 125 figures; 11 tables; 5 appendices; the total number of pages is 208, including appendices. The Bibliography contains 122 titles.

TABLE OF CONTENTS

INTRODUCTION

While software engineering evolves, it includes more disciplines. Nowadays, these disciplines include development, business process analysis, project management and others. Authors of [1] consider modern software engineering to be a result of Agile methodology and Model-Driven Architecture (MDA) development. One of the newest software engineering paradigms is model-driven software development (MDSD) [18], [75]. In contrast to so called "model-based" development, that uses different kinds of models in different development stages, MDSD is based around strictly defined models and its processing algorithms.

It is possible to define two kinds of MDSD users [75] – ones that consider the model to be purely analytical instrument meant for problem domain analysis and business modelling, as well as others, that define model as high-level executable artifact. Such an artifact can in turn be used to automatically obtain lower-level artifacts (software code, documentation, and others). So, it is possible to say, that a model should be both understandable to all the stakeholders as well as fitted for an automatic code generation.

Unified modelling language (UML) [90] is widely used by enterprises [29]. While it supports code generation from standard models (for example, [6], [7], [18], [79], [83], [91]), it can be explained by the fact that UML diagrams are used to describe not the initial problem domain, but existing solutions. OMG (Object Management Group) defines UML as "a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems" [90]. This, in turn, means that UML is not meant for business analysis and should be used to describe solution architecture.

Similar thoughts are expressed by many authors, for example in [8], [74], [85]. These authors are pointing out the fact that modern modelling languages (including UML) can be understood by model-driven engineering experts, while being unclear for problem domain experts and business analysts. An idea that UML models are "too similar to the software code to be used in MDSD" is brought out. So, it is necessary to have a modelling language understandable by all stakeholders. The following artifacts are usually defined as fitting in this role:

- use case list;
- business process models describing aforementioned use cases (for example, BPMN – Business Process Model and Notation [10]);
- concept models containing information on problem domain objects.

So, it is possible to note that code generation from models is possible, but currently used models are not always suited for the task due to their complexity. Author of this Thesis states a hypothesis that modelling language should consist of two artifacts (similar to [8]):

- business process model, which can be BPMN [10], data flow diagram (DFD) [25], [82] or others;
- conceptual domain model, which can be, for example, represented in the entity-relationship (ER) [17] diagram form.

Use case model can be fully replaced by the business process model – by defining business process flow for each use case, same information is obtained. This set of artifacts should prove usable for wider MDSD introduction.

Confirmation of the Proposed Model Hypothesis

To prove the mentioned hypothesis, two surveys were conducted. The first one targeted software developers and architects and asked a single question about UML diagrams that respondents can define. A total of 227 answers was received, and analysis of these answers show that the majority of respondents can define class, sequence, activity and use case diagrams. This can be explained by the fact that these diagrams (excluding use case) are used to describe system architecture and process flow.

The second survey was meant for business analysts. In addition to the previous question respondents were asked about their experience with business process and ER-diagrams. A total of 46 answers was received, and analysis of these answers proves that the **majority of business analysts can understand, define, and use business process and ER-diagrams.**

The survey results prove that the business process and ER models should probably improve the situation with wide MDSD adoption. So, it is possible to conclude that the code generation task would require modelling language that supports these models.

As such model a two-hemisphere model was selected [54], [64]. Several researches, that were performed, show that it is possible to use this model for UML diagram generation, since the model itself contains enough information to define both static and dynamic artifacts [54], [55]. The transformation result is an UML diagram set that can be further used to generate a code [71]. So, another hypothesis is defined: **if it is possible to perform two-hemisphere model → UML and UML → code transformations, then it should also be possible to transform a two-hemisphere model into software code directly**. This fact is also confirmed in [33], [34].

Such transformation would allow **to use a single model for the code generation task** – two-hemisphere model, which, in turn, is a result of business requirement analysis. This means, that in the future it would only be necessary to support this model development to provide a full MDSD solution to all the stakeholders.

Purpose and Tasks the Doctoral Thesis

The purpose of this Thesis is to develop a code generation algorithm that uses two-hemisphere model as a source model, validation method for this algorithm, as well as perform a practical approbation of the developed algorithm. To achieve this, it is necessary to perform the following tasks.

1. To evaluate the two-hemisphere model usability in the code generation context and improve the model, if necessary.
2. To select the programming language for the code generation and substantiate the choice.

3. To develop an algorithm for the two-hemisphere model transformation into a software code.
4. To define a validation methodology to check the correctness of transformation.
5. To evaluate the developed algorithm in practice by using it to develop a software system.

Scientific Innovation and Practical Significance of the Doctoral Thesis

Scientific innovation of the Doctoral Thesis is as follows

1. An analysis of two-hemisphere model was performed. Both its advantages and limitations in software code generation context were identified. After this analysis, several improvements were defined for the model.
2. New transformation rules were developed to support a code generation from the two-hemisphere model. In this Thesis, Java code generation programming language [41] is described, however, it is possible to use the proposed rules to obtain the code in different programming languages.
3. To make transformation rules language-agnostic, intermediate model to represent the code was developed. This model supports code generation for different paradigm languages.
4. An improved class relationship detection algorithm was developed. It can be used also outside of the model-driven engineering scope. Examples of such usage would be refactoring, existing system and its component analysis and others.

Practical significance of the research

Results that were obtained during Thesis development were also used to solve a practical task – microcontroller software development. During this development the author noted that the two-hemisphere model is understandable not only to the software engineers but to other stakeholders as well, which helped in improving the communication. The defined transformation rules are language-agnostic and can be used with different paradigm programming languages, which was also proven during the practical approbation.

Thesis Approbation

The results of this Thesis are reflected in 9 publications in international and Latvian Council of Science recognized scientific issues

1. Nikiforova, O., Gusarovs, K. Anemic Domain Model vs Rich Domain Model to Improve the Two-Hemisphere Model-Driven Approach. *Applied Computer Systems*, 2020, Volume 25, Issue 1. (Accepted for publication). (Contribution ~50 %). †
2. Gusarovs, K., Nikiforova, O. An Intermediate Model for the Code Generation from the Two-Hemisphere Model. In: *Proceedings of the 14th International Conference on Software Engineering Advances (ICSEA 2019)*, accepted for publishing. ISBN: 978-1-61208-752-8. (Contribution ~70%).

3. Gusarovs, K. An Analysis on Java Programming Language Decompiler Capabilities. *Applied Computer Systems*, 2018, Volume 23, No. 2. pp. 109–117. eISSN 2255-8691. ISSN 2255-8683. Available from: doi:10.2478/acss-2018-0014.

4. Gusarovs, K., Nikiforova, O., Giurca, A. Simplified Lisp Code Generation from the Two-hemisphere Model. *Procedia Computer Science*, 2017, Volume 104, Issue C. pp. 329–337. Available from: doi:10.1016/j.procs.2017.01.142. (Contribution ~70 %). *†

5. Gusarovs, K., Nikiforova, O. Workflow Generation from the Two-Hemisphere Model. *Applied Computer Systems*, 2017, Volume 22, Issue 1. pp. 36–46. eISSN 2255-8691. ISSN 2255-8683. Available from: doi:10.1515/acss-2017-0016. (Contribution ~60 %). †

6. Nikiforova, O., Gusarovs, K., Ressin, A. An Approach to Generation of the UML Sequence Diagram from the Two-Hemisphere Model. In: *Proceedings of the 11th International Conference on Software Engineering Advances (ICSEA 2016)*. Wilmington: IARIA, 2016, pp. 142–149. ISBN 978-1-61208-498-5. (Contribution ~40 %).

7. Gusarovs, K., Nikiforova, O., Jukss, M. A Prototype of Description Language for the Two-Hemisphere Model. *Applied Computer Systems*, 2015, Volume 18, Issue 1. pp. 15–20. ISSN 2255-8691. Available from: doi:10.1515/acss-2015-0014. (Contribution ~60 %).

8. Nikiforova, O., Gusarovs, K. Kozacenko, L., Ahilcenoka, D., Ungurs, D. An Approach to Compare UML Class Diagrams Based on Semantical Features of Their Elements. In: *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA 2015)*. Wilmington: IARIA, 2015, pp. 147–153. ISBN 978-1-61208-438-1. (Contribution ~40 %).

9. Zusane, U.I., Nikiforova, O., Gusarovs, K. Several Issues on the Model Interchange Between Model-Driven Software Development Tools. In: *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA 2015)*. Wilmington: IARIA, 2015, pp. 451–457. ISBN 978-1-61208-438-1. (Contribution ~20 %).

The results of the Thesis were presented in 7 international scientific conferences:

1. ICSEA 2015 – The Tenth International Conference on Software Engineering Advances, November 15–20, 2015, Barcelona, Spain, "Several Issues on the Model Interchange Between Model-Driven Software Development Tools" and "An Approach to Compare UML Class Diagrams Based on Semantical Features of Their Elements".

2. Riga Technical University 56th International Scientific Conference, October 14–17, 2015, Riga, Latvia, "A Prototype of Description Language for the Two-Hemisphere Model".

3. ICSEA 2016 – The Eleventh International Conference on Software Engineering Advances. August 21–25, 2016, Rome, Italy. "An Approach to Generation of the UML Sequence Diagram from the Two-Hemisphere Model".

4. Riga Technical University 58th International Scientific Conference, October 12–15, 2017, Riga, Latvia, "Workflow Generation from the Two-Hemisphere Model".

5. 15th International Conference of Numerical Analysis and Applied Mathematics ICNAAM 2017; 7th Symposium on Computer Languages, Implementation and Tools – SCLIT 2017, September 26–30, 2017, Thessaloniki, Greece, "Several Issues of Two-Hemisphere Model-Driven Approach to Improve with Anemic Domain Model".

6. Riga Technical University 59th International Scientific Conference, October 10–12, 2018, Riga, Latvia, "An Analysis on Java Programming Language Decompiler Capabilities".

7. ICSEA 2019 – The Fourteenth International Conference on Software Engineering Advances, November 24–28, 2019, Valencia, Spain, "An Intermediate Model for the Code Generation from the Two-Hemisphere Model".

Other publications on the research topic:

1. Nikiforova, O., Ahilcenoka, D., Ungurs, D., Gusarovs, K., Kozacenko, L. Several Issues on the Layout of the UML Sequence and Class Diagram. In: *Proceedings of the 9th International Conference on Software Engineering Advances (ICSEA 2014),* 2014, pp. 40–47. (Contribution ~20 %).

2. Nikiforova, O., Bohomaz, Y., Gusarovs, K. A Comparison of the Implementation Means for Development of Modelling Tool. In: *Proceedings of the 2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS 2017),* 2017, pp. 1–6. Available from: doi:10.1109/WITS.2017.7934623. (Contribution ~20 %). *†

3. Nikiforova, O., El Marzouki, N., Gusarovs, K., Vangheluwe, H., Bures, T., Al-Ali, R., Iacono, M., Esquivel, P.O., Leon, F. The Two-Hemisphere Modelling Approach to the Composition of Cyber-Physical Systems. In: *In Proceedings of the 12th International Conference on Software Technologies.* Portugal: SciTePress, 2017, pp. 286–293. ISBN 978-989-758-262-2. Available from: doi:10.5220/0006424902860293. (Contribution ~10 %). *†

4. Nikiforova, O., Gorbiks, O., Gusarovs, K., Ahilcenoka, D., Bajovs, A., Kozacenko, L., Skindere, N., Ungurs, D. Development of BrainTool for Generation of UML Diagrams from the Two-hemisphere Model Based on the Two-Hemisphere Model Transformation Itself. In: *Proceedings of the International Scientific Conference "Applied Information and Communication Technologies".* Latvia: LLU, 2013, pp. 267–274. ISSN 2255-8586. (Contribution ~20 %).

5. Nikiforova, O., Gusarovs, K. Comparison of BrainTool to Other UML Modeling and Model Transformation Tools. *AIP Conference Proceedings*, 2017, Volume 1863, No. 1, id. 330005. Available from: doi:10.1063/1.4992503. (Contribution ~40 %). *†

6. Nikiforova, O., Gusarovs, K., Gorbiks, O., Pavlova, N. BrainTool. A Tool for Generation of the UML Class Diagrams. In: *Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA 2012),* 2012. Lisbon: IARIA, 2012, pp. 60–69. ISBN 9781612082301. (Contribution ~35 %).

7. Nikiforova, O., Gusarovs, K., Gorbiks, O., Pavlova, N. Improvement of the Two-Hemisphere Model-Driven Approach for Generation of the UML Class Diagram. *Applied Computer Systems*, 2013, Volume 14, Issue 1, pp. 19–30. ISSN 2255-8691. Available from: doi:10.2478/acss-2013-0003. (Contribution ~30 %).

8. Nikiforova, O., Kozacenko, L., Ungurs, D., Ahilcenoka, D., Bajovs, A., Skindere, N., Gusarovs, K., Jukss, M. BrainTool v2.0 for Software Modeling in UML. *Applied Computer Systems*, 2015, Volume 16, Issue 1, pp 33–42. ISSN 2255-8691. Available from: doi:10.1515/acss-2014-0011. (Contribution ~10 %).

9. Nikiforova, O., Kozacenko, L., Ahilcenoka, D., Gusarovs, K., Ungurs, D., Jukss, M. Comparison of the Two-Hemisphere Model-Driven Approach to Other Methods for Model-Driven Software Development. *Applied Computer Systems*, 2016, Volume 18, Issue 1, pp. 5–14. ISSN 2255-8691. Available from: doi:10.1515/acss-2015-0013. (Contribution ~20 %).

10. Nikiforova, O., Pavlova, N., Gusarovs, K., Gorbiks, O., Vorotilovs, J., Zaharovs, A., Umanovskis, D., Sejans, J. Development of the Tool for Transformation of the Two-Hemisphere Model to the UML Class Diagram: Technical Solutions and Lessons Learned. In: *Proceedings of the 5th Internation Scientific Conference "Applied Information and Communication Technology 2012"*. Latvia: LLU, 2012, pp. 11–19. ISBN 978-9984-48-065-7. (Contribution ~30 %).

11. Nikiforova, O., Sukovskis, U., Gusarovs, K. Application of the Two-Hemisphere Model Supported by BrainTool: Football Game Simulation. *AIP Conference Proceedings*, 2015, Volume 1648, No. 1, id. 310004. Available from: doi:10.1063/1.4912557. (Contribution ~25 %). †

* indexed in SCOPUS, † indexed in Web of Science.

Structure of the Doctoral Thesis

The Thesis consists of introduction, seven sections, conclusion, bibliography and five appendices.

Introduction justifies the topicality of the chosen theme, defines the purposes and tasks of the Thesis to accomplish that purpose.

Section 1 covers the two-hemisphere model and provides a short insight into actual transformation methods that can be used to obtain different artifacts from it.

In Section 2, limitations in code generation context of the existing two-hemisphere model notation and its transformation rules are analysed. Solutions to the identified limitations are described.

In Section 3, the development of domain-specific language that can be used to define a two-hemisphere model is described. The need for such a language is due to necessity to have an easy modifiable two-hemisphere model notation.

An improved class relationship identification algorithm is described in Section 4. This algorithm is meant for class set processing. Such an algorithm is needed due to the limitations of one of the identified transformation rules.

The developed transformation rules are described in Section 5. In the beginning, target programming language is selected and intermediate model to be used during transformations is defined. Then transformation rules to obtain such a model are described.

In Section 6, the process of intermediate model transformation to Java code is described. To do this, the author defines an example model that contains cases of special tests. Transformation result validation is also described in this section along with related work analysis.

Practical approbation of transformation rules is described in Section 7. The system to be developed is described, its two-hemisphere model is defined, and necessary modifications to code generation algorithm are noted. Finally, the analysis of obtained results is provided.

Conclusions provide an insight into the research results.

The Thesis has five appendices. Appendix 1 contains the results of software developer and architect survey. Appendix 2 contains business analyst's survey results. Appendix 3 contains the notation of DSL for the two-hemisphere model description. Appendix 4 contains the example two-hemisphere model that was used to verify the developed transformation rules, while Appendix 6 contains the source code of microcontroller software that was obtained during the practical approbation of the proposed method.

1. TWO-HEMISPHERE MODEL AND ACTUAL TRANSFORMATION RULES

The name "two-hemisphere model" is chosen based on cognitive psychology view [2], when the human brain is considered to have two hemispheres. One of the hemispheres is responsible for logic, the second – for concepts. For a human being to function normally, coordinated operation of both hemispheres is necessary. Similar principle is applied in the two-hemisphere model. It consists of two diagrams – business process diagram on the left side of the model and concept diagram on the right. The concept diagram describes data that is used in the model.

Both diagrams are interconnected. It is done by linking concepts from the concept diagram to data flows in the business process diagram. Such linking allows not only for model interconnection, but also ensures that each and every data flow has a strictly defined data type [58]. It is also possible to define the business process performers, that can be system users, other systems, or even abstract concepts such as a database.
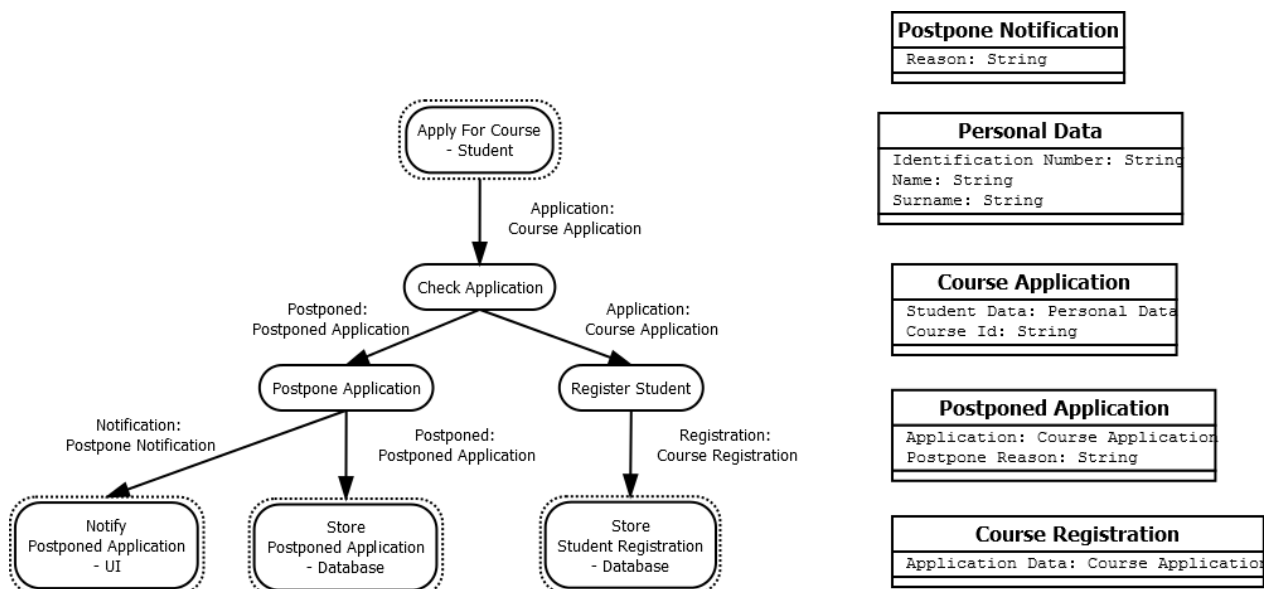

Fig. 1.1. An example of a two-hemisphere model.

Figure 1.1 depicts an example of a two-hemisphere model. Here, a process of student applying for a course is described. In the beginning, it is necessary to validate the received application – if it is not correct, an error message is returned. Then there is checking for the possibility to enlist a student (for example, due to the competition). If this is not possible, the application is stored in the postponed application database to ensure that the student can be notified if there is a possibility to join the course. Also, a notification to the student is fired. Otherwise, the student is enlisted to a course and his personal information is stored in the enlisted student database.

By the time the development of Doctoral Thesis started, several two-hemisphere model transformation methods, allowing to obtain different artifacts, have been defined.

- Research [68] describes a method for UML [90] class diagram generation from a two-hemisphere model. For the transformation purposes the source model is converted into intermediate model, which in turn is used to generate a communication diagram. This diagram serves as a source model for the resulting class diagram.

- Research [62] offers several improvements to the previous transformation method with a goal to obtain a more precise class diagram and class specifications. To do this, the research authors propose to introduce an additional transformation step, which is meant for obtaining additional information. This step is semi-automatic, and its output is so-called Matrix of the Required Transitions.

- Work [65] describes the UML sequence diagram generation from the two-hemisphere model. The proposed method is based on the analysis of the business process model structure. As a result of transformation, such elements of sequence diagram as actors, objects, messages, and parallel execution fragments, are defined.

- In [63] the authors for the first time propose an alternative approach to the two-hemisphere model transformation. The business process diagram is compared to a finite-state machine (FSM) [47], [80], [96], and transformations are based on the fact that any FSM can be represented as a regular expression. As a result of this conversion, the UML sequence diagram containing actors, objects, messages, loops, and alternative execution fragments are defined.

- Research [33] is an improvement of the previous transformation method that aims at code generation directly from the two-hemisphere model (without using UML diagrams). However, its authors acknowledge several limitations of the proposed method.

- This Doctoral Thesis is based on [34]. The ideas described in [33] and [63] (business process model comparison to the FSM and its processing using specialized algorithms) are developed further.

2. LIMITATIONS OF TWO-HEMISPHERE MODEL FOR CODE GENERATION TASK

By the time when the development of the Doctoral Thesis was started multiple two-hemisphere model transformation methods have already been defined. These methods allowed mainly for a static information (classes, class attributes, methods, and relations) generation [66]. It is necessary to note that even the proposed method of UML sequence diagram generation [65] is not complete – the authors acknowledge that the method is incomplete for sequence diagram fragment generation. So, it is possible to conclude, that the existing transformation methods are limited to static artifact production, which in turn means that part of the information contained in the source model is not used.

Since the task of this Thesis is code generation from a two-hemisphere model, it is necessary to analyse the target model to understand what information in addition to already contained in the source model, software code has. It is also necessary to understand if at least part of this information must be appended to the two-hemisphere model notation. If it is so, it is possible to conclude, that there is a limitation in the current notation to be mitigated.

Software code [38] consists of two main parts.

1. Declarations – programming language constructions, that define one or more identifiers and ways of interpreting these identifiers.
2. Instructions (operations) that define executable actions, their operands, and results. Sequence of operation execution defines one or more algorithms, that allow software code to produce necessary results.

The existing rules of the two-hemisphere model transformation allow to turn the concept model into a set of classes, which now is enough to define declarations. However, the generation of instruction (operation) sequence is done only in the form of UML sequence diagram [65]. To produce this information, the existing methods use the initial business process model. A simple instruction sequence (without branching) can be generated using the current notation of the two-hemisphere model.

Further analysing the existing transformation methods, it is possible to note that transformation results are presented in the so-called rich data model form. The main task of any software system can always be reduced to data processing. Software is responsible for data input, processing, and data output that can be done in different ways. Object-oriented paradigm is based on software component definition in a form of objects. Data is also part of software, so object-oriented approach means data definition in object form as well. The evolution of object-oriented approach resulted in two data model representations [13], [86] – anaemic [21], [23] and rich data modes. The rich data model is based on data and its processing logic composition, while the anaemic approach is all about separation of concepts – data and logic are different software components that should be also represented as such. It is also necessary to note, that the anaemic approach allows easier definition and support for Model-View-Controller (MVC) [89] architecture.

Considering both thoughts expressed by different authors in [21], [23], [49], and the author's own experience in widely used frameworks (for example, Hibernate [35], AngularJS

[3], and others), the author of the Thesis concludes, that the anaemic model also has to be supported. This is also proven in [86]. It is possible to say, that data (domain objects) in real life examples almost never contain any kind of logic – they are meant to represent database tables and data transfer objects (input/output). Therefore, the following can be concluded.

1. Inheritance of domain objects is meant only for an attribute set extension.
2. Polymorphism is not a common case for data – it is possible to separate different types of data by adding necessary attributes with the help of inheritance.
3. Usually data processing services are meant to support concrete data types, which minimizes the necessity for abstraction, encapsulation, and polymorphism in domain objects.
4. Multiple existing data processing frameworks and libraries (for example, widely used object-relation mapping framework Hibernate [35]) use the anaemic data model.

So, it is considered, that the anaemic data model is not an antipattern, and its adoption is perfectly permissible in MDSD.

It is possible to see, that both two-hemisphere model notation as well as its transformation rules have limitations, which can make it difficult or impossible to generate code. Table 2.1 provides a short summary of identified limitations along with mitigation possibilities.

Table. 2.1

Existing Two-Hemisphere Model and Transformation Rule Limitations

Limitation	Limitation in	Mitigation possibility
It is not possible to define preconditions of process execution	Notation	Enrich notations with elements that contain such information
In case of multiple outgoing data flows it is not possible to define if the process produces one or multiple of these	Notation	It is possible to add information to each outgoing data flow in order to define if it is always produced because of the process execution. It is also possible to create an object that contains all outgoing data flows in the form of its attributes
Data assigned to data flows is limited to a single concept	Notation	Offer a possibility for data flows to have primitive data types, arrays/collections, or even no data assigned
After the class set is generated, it is never analysed. As a result, it is possible to have repetitions in the resulting code. Also, there is a possibility that not all class relations are defined	Transformation rules	Define an algorithm for the class set processing to define more precise class relations and get rid of repetitions
Transformation rules do not support the anaemic data model generation [60]	Transformation rules	Support anaemic data model in code generation
Sequence diagram generation algorithm does not allow for different fragment type definition, as well as preconditions	Transformation rules	Enrich both model notation and transformation rules to support these requirements

3. DOMAIN-SPECIFIC LANGUAGE FOR TWO-HEMISPHERE MODEL DEFINITION

To improve the limitations identified in the two-hemisphere model notation, it is necessary to modify it. Previous research on a two-hemisphere model and its transformations was based around two tools that support existing graphical notation BrainTool [61] and BrainTool 2.0 [66]. While these tools offer transparent model visualisation, its modifications might be costly in case of adding new elements [58], [69]. Therefore, in this Thesis it is offered to define a domain-specific language (DSL) to represent the two-hemisphere model [31].

In contrast to general purpose languages (GPL), domain-specific languages are meant for exact problem solving or transparent and precise definition of a concrete business domain information [9], [51]. The universality of such languages is diminished in favour of expressivity in concrete domain.

To define a domain-specific language, it is necessary to identify the main elements of a problem domain and create its representations using the selected syntax. It is possible to use different notations for a DSL definition, for example, XML schema [93] or extended Backus–Naur form (EBNF) [28].

In the case of MDSD domain specific language is often considered to be a graphical model [9], [11], [44], [51], [92]. This fact can probably be explained by vast amount of purely theoretical research in this area. Only a few of proposed methods are supported by tools that allow for an appropriate model definition and transformations – development of such a tool can be costly [61], [66] and impose limits on experimentation possibilities [56], [58], [61], [69].

Table. 3.1

Limitations of Two-Hemisphere Model Notation Solved by DSL

Limitation	Solution using DSL
It is not possible to define the process execution preconditions	Model notation is enriched with additional elements that allow to define the execution precondition of a business process
In the case of multiple outgoing data flows it is not possible to define if the process produces one or multiple of these	This limitation is not solved using DSL – it is possible to mitigate the solution to the transformations. Since it is possible to generate the code in different programming languages, it is possible that some of these languages (for example, Python [95]) allow for multiple results
Data assigned to data flows is limited to a single concept	In DSL it is possible to assign primitive data types, objects, arrays, or data collections to the data flow, as well as leave it without assigned data

The syntax of the developed language is described using EBNF, so it is possible to use special tools (such as [4], [87]) for the creation of its parser. In the Thesis, ANTLR [4] tool that generates Java [41] code is chosen to perform this task. The developed domain-specific language not only allows to define the two-hemisphere model, but also enriches its notation with new elements and fixes the identified limitations [31]. Short summary on how this is achieved is provided in Table 3.1.

A simple two-hemisphere model defined using both graphical and DSL notations is shown in Figs. 3.1 and 3.2. It is possible to conclude that the definition that is done using DSL contains additional information, which in turn can be used during the model transformation.
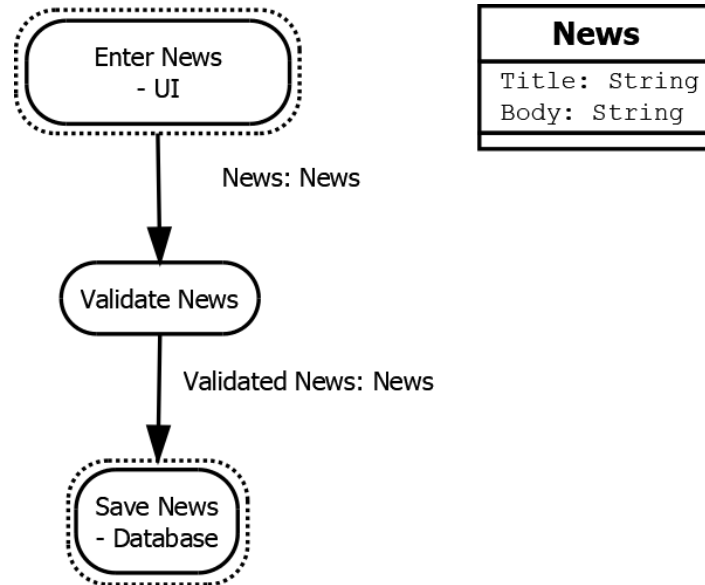

Fig. 3.1. Graphical notation of a simple two-hemisphere model.

```
DEFINE CONCEPT News WITH NAME "News"
    AND ATTRIBUTE "Title"(String)
    AND ATTRIBUTE "Body"(String)
END CONCEPT
DEFINE PROCESS EnterNews WITH NAME "Enter News"
    AND PERFORMER "UI"
    AND TYPE EXTERNAL
END PROCESS
DEFINE PROCESS ValidateNews WITH NAME "Validate News"
END PROCESS
DEFINE PROCESS SaveNews WITH GUARD "News are valid" AND NAME "Save News"
    AND PERFORMER "Database"
    AND TYPE EXTERNAL
END PROCESS
DEFINE DATA FLOW EnteredNews FROM EnterNews TO ValidateNews WITH NAME "News"
    AND PARAMETER "n"(News)
END DATA FLOW
DEFINE DATA FLOW ValidatedNews FROM ValidateNews TO SaveNews WITH NAME "Validated
News"
    AND PARAMETER "n"(News)
    AND PARAMETER "Validation Result"(Boolean)
END DATA FLOW
DEFINE PROCESS MODEL ValidateAndSaveNews WITH NAME "Validate and save news"
    AND PROCESS EnterNews
    AND PROCESS ValidateNews
    AND PROCESS SaveNews
END PROCESS MODEL
```

Fig. 3.2. DSL notation of a simple two-hemisphere model.

4. ALGORITHM FOR IMPROVED CLASS RELATIONSHIP DETECTION

Researches presented in [55], [57], [59], [67], [69], [70], and others prove that it is possible to create transformation rules that define not only classes but also their relationships. However, the existing transformation rules define class relationships only by using information from the business process model. While this approach is correct, since classes that interoperate should be connected by dependency or association relationship [90], other relationships (aggregation, realization, generalization) can also require the analysis of generated class set. For example, during transformation several classes that share similar attributes can be generated. Such classes should become parts of a single class hierarchy [50]. In a similar way aggregation detection does not rely on information that is already contained in the conceptual model. So, the author defines such requirements for the class relationship detection algorithm.

1. It should be able to process a class set that can be represented in various ways – for example, source code classes, or set of UML classes, or differently.
2. The algorithm should use information on class interoperation and analyse class structure – their attributes and methods.
3. The input of algorithm should contain information on class interoperation as an additional parameter. This is required to make the results of this algorithm more precise and rely on information contained in the initial business process model as well.
4. The names of classes and interfaces defined by algorithm should be provided by a human instead of being generated automatically.

The developed algorithm consists of four main steps. During the first step generalization and realization relationships are defined. The second step is responsible for aggregations, the third step – for dependencies, while the last one defines associations. The input of algorithm is a set of classes (where each class has information on its attributes and methods) as well as information on class interoperation. The output of algorithm is a class set enriched with class relationships.

During the generalization and realization detection human involvement is necessary. This means that this step is performed in a semi-automatic way. In the beginning the input classes are analysed to construct the so-called common element table. This table contains information on class methods and attributes and allows for class structure comparison and analysis. After such a table is defined, a generalization detection loop is started. During each iteration of this loop two classes – A and B that share most common elements, are selected from a common element table. After that, human analyst is offered four choices.

1. Convert class A to base class, B – to derived.
2. Convert class B to base class, A – to derived.
3. Define new base class S, classes A and B convert to derived from S.
4. Do nothing.

In accordance to the selected action, the resulting class set is enriched with the defined class relationships, as well as new classes, if such are defined.

Realization definitions are performed in a similar manner. A hash table, where key is method, value – set of classes that share this method, is defined. By analysing this table, the algorithm can offer to define a new interface. In the case when the analyst chooses to proceed with definition, the algorithm checks if an interface that is implemented by all the candidate classes already exists. If such an interface is not found, a new one is defined and added to the result class set. After an appropriate interface has been identified, it is being enriched with a method common for all classes. Then, this method is removed from the appropriate class definitions.

Next type of class relationships that is defined by the proposed algorithm is aggregation. It is possible to analyse a class set with a goal to identify situations when one class contains the other in a form of its attribute. It is also necessary to consider situations when class A, which is a base class for class B, contains B as an attribute. In such cases aggregation is not defined, since generalization (or realization) relationship is already defined for these two classes. In a similar way it is possible that class B, that is derived from class A, contains A as an attribute. Once again, in such a case aggregation relationship will not be defined.

To define dependency relationships both class method definitions and information on class interoperation is utilised. So, this step of algorithm consists of two sub-steps. During the first sub-step method parameters and their return types are being analysed. After that the algorithm checks information on class interoperation. When it is possible to define the dependency relationship, it is necessary to check if there is no previously defined relationship between appropriate classes. Generalization, realization, and aggregations relationships are more "important" than dependency.

During association detection already identified dependencies are analysed. If several dependencies exist between two classes, these dependencies are replaced with association relationship.

In the Doctoral Thesis an example of algorithm operation is provided. In this example input is a class set that contains no class relationships. As a result of algorithm execution, a fully connected class set is obtained. The developed class relationship detection algorithm can be used not only for the two-hemisphere model transformation but also serve other purposes – for example, refactoring [24]. It is also possible to use it alongside other model-driven approaches.

5. CODE GENERATION FROM TWO-HEMISPHERE MODEL

5.1. Selection of Target Programming Language

To select a target programming language both information on its usage [88] as well as requirements for it to be strictly-typed and general-purpose language were defined. An example of widely used non-GPL language is SQL [39]. Cross platform capabilities and garbage collected memory management were also considered. As a result, Java [41] was selected as a target language.

5.2. Code Generation Strategy Definition

It is offered to perform the two-hemisphere model transformation process in several steps.
1. In the beginning it is necessary to define resulting data structures.
2. Then information from the business process model is processed to obtain the signatures of methods (or functions being generated).
3. Using previously created definitions it is possible to define a so-called workflow that describes the execution sequence and preconditions for the appropriate methods/functions.
4. Generation of the source code.

To support such a process, an intermediate model is used [32]. After execution of steps 1–3 a representation of source code that can be converted to the selected programming language is obtained. By doing so, it is possible to modify only the last transformation step for the new target programming language support. The business process model is converted to a workflow that is defined using an assembler-like [78] language (described in Table. 5.1). Figure 5.1 provides an example of Java code, while Fig. 5.2 contains the same code fragment, but defined using the proposed approach.

Table. 5.1

Intermediate Model Instructions

Instruction	Description	Examples
`Label<Name>`	Label placement	`Label<1>`
`Var<Id, Name, Type>`	Variable definition	`Var<1, S, String>`
`Invoke<Method, Inputs, Output?>`	Method/process execution	`Invoke<fn, [a, b, c], d>` `Invoke<doNothing, []>`
`GetField<Object, Field, Target>`	Storing object attribute into a variable	`GetField<Obj, left, x>`
`PutField<Object, Field, Source>`	Storing variable into an object attribute	`PutField<Obj, left, x>`
`Jump<Label>`	Unconditional branching	`Jump<Label1>`
`Check<Var, Guard>`	Condition check	`Check<cond, "X > 0">`
`JumpIf<Var, Label>` `JumpIfNot<Var, Label>`	Conditional branching	`JumpIf< cond, Label1>` `JumpIfNot<cond, Label2>`
`Return<Var?>`	Return of a method/process execution result	`Return` `Return<x>`

```
int i = rand();
while (i > 0) {
    System.out.println(i);
    i -= 2;
}
```

Fig. 5.1. Java code example.

```
Var<1, i, int>
Var<2, b, boolean>
Invoke<rand, [], 1>
Label<L1>
Check<2, "i > 0">
JumpIf<2, L2>
Invoke<System.out.println, [i], ∅>
Label<L2>
Return<∅>
```

Fig. 5.2. Source code representation.

5.3. Minimization of a Business Process Diagram

To be able to transform the business process diagram (which is an oriented multigraph with circles [20]) to a sequence of instructions, it is necessary to process it by changing the graph structure. Research [63] offers to consider the business process diagram graph to be a finite state machine [47], [80], [96] to support such a transformation. This allows to apply algorithms meant for FSM processing to the business process diagram.

Each business process from the business process diagram is converted into a so-called method signature shown in Fig. 5.3. This signature consists of a process name, information on incoming and outgoing data flows, as well as process execution precondition.

```
class MethodSignature {
    const BusinessProcess process;
    const Set<DataFlow> parameters;
    const Set<DataFlow> outputs;
    String guard;
}
```

Fig. 5.3. Method signature definition with execution precondition.

After the method signatures are defined, it is possible to perform the first transformation of the business process diagram. As a result, a so-called process invocation graph is generated. Such a graph is similar to the functional model described in [27]. It also corresponds to the source process diagram with a few modifications.

1. Edges of a graph are converted to the connecting elements that are used to define all the possible process execution sequences. They do not correspond to data flows any more. Business processes, in turn, are replaced with appropriate method signatures.
2. Initial and terminal vertex that define according system states, are appended to a graph. All the external processes are connected to these vertices.

Figure 5.4 contains a fragment of the process invocation graph that describes two sequential process executions.

1. `P1` receives `a` and `b` as parameters and returns `c` as a result.
2. `P2` receives `c` as a parameter returning `d` and `e` as a result.
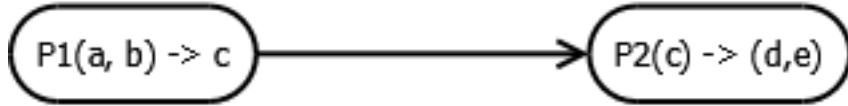

Fig. 5.4. Process invocation graph fragment.

These two processes are connected with a single edge, which in turn means, that only one execution sequence is possible − `P1` → `P2`. In this case it is possible to merge both vertices into a new one, that will contain both process executions. Such a vertex is shown in Fig. 5.5.
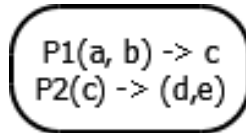

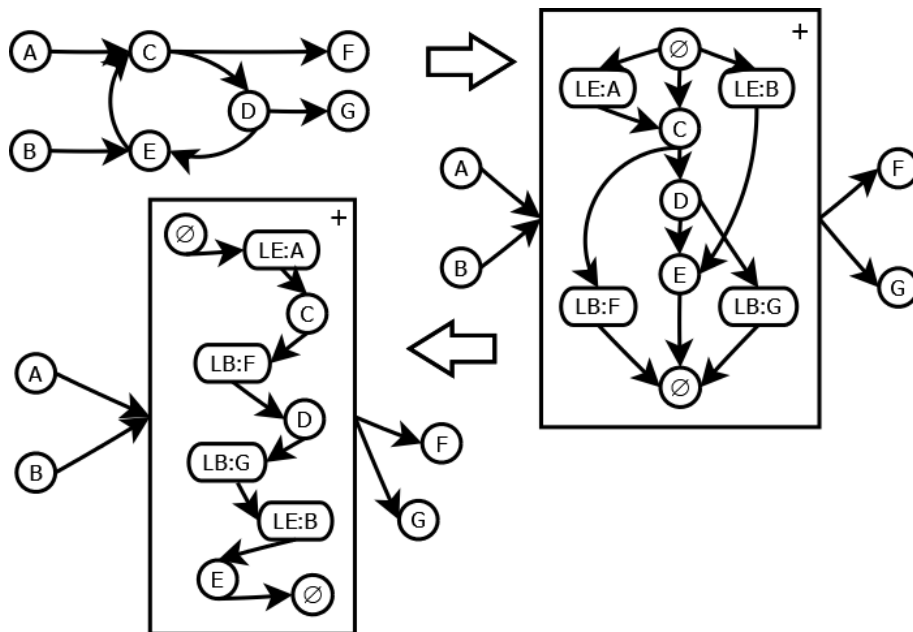Fig. 5.5. Result of vertex merge.

Fig. 5.6. Loop replacement.

It is possible to see that by merging two vertices a new one, containing information about two processes and its execution sequence, was created. Such merging allows both to reduce the number of vertices and graph as well as include information on multiple processes into a single vertex.

Other possible ways to merge the information on multiple processes were identified. By applying these merges it is possible to reduce the graph to a state when it consists of only one vertex. This vertex, in turn, will contain all the information present in graph initially. The content of such a vertex is a linear structure that corresponds to the target model − source

code. So, it is necessary to define multiple graph processing algorithms that modify its structure keeping the information contained. It is possible to define two kinds of such algorithms.

The first type of graph minimization algorithm is meant for loop processing. Application of these algorithms is shown in Fig. 5.6. In the beginning, all the loops in the graph are identified by using Tarjan's [84] and Johnson's [43] algorithms. Then each loop is replaced with a new process invocation graph that contains all the replaced vertices. Two additional vertex types are defined:

- `LE: from` − loop entry point (loop execution can begin from this vertex);
- `LB: to` − loop exit point (it is possible to break the loop execution from this vertex).

After loop replacement it is possible to transform each of the new graphs and put the transformation results back into the initial graph. When doing this, it is necessary to consider some additional rules.

1. If there are multiple paths between vertices `A` and `B`, some of which contain only loop entry or exit points, then it is possible to merge these paths by creating a new subgraph that will contain these vertices as a sequence. This subgraph is then placed before vertex `B`. (Fig. 5.7).
2. By analysing vertices before and after the loop, it is possible to reorder and possibly remove redundant loop entry and exit points. Also, multiple identical sequential entry/exist points can be reduced to one (Fig. 5.8).
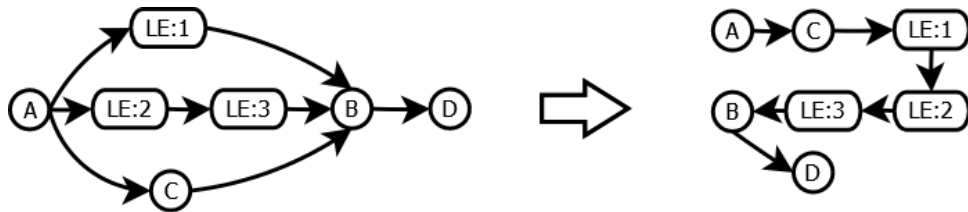

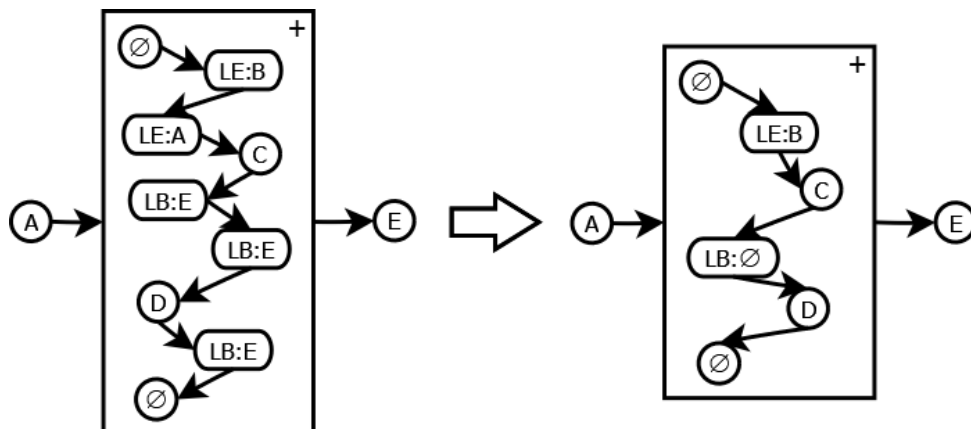Fig. 5.7. Processing of alternative loop entry and exit points.

Fig. 5.8. Loop entry and exit point processing.

Other processing algorithms, that are defined in the Doctoral Thesis, are presented in Table 5.2.

Process Invocation Graph Transformations

Name	Transformation
Circle replacement	
Removal of duplicate edges	
Sequential vertex merging	
Disjunction definition	
Loop condition definition	
Sequence condition definition	

To verify the defined graph processing algorithms, a total of 10 000 experiments were conducted. During these experiments randomly generated graphs consisting of different vertex count (from 10 to 100) were processed by the proposed algorithm set. Such graphs were generated with several additional requirements:

- graph had to contain at least one circle;
- graph had to contain at least one cycle;
- at least 10 % of the graph vertices had to be connected with multiple other vertices – both by incoming and outcoming edges;
- graph had to contain at least one start and one end vertex (that denote the first and last processes). In addition, each of the end vertices had to be reachable from at least one of the start vertices. Also, between each start and end vertex at least one path had to exist;
- graph had to be weakly connected.

All the graphs generated during the experimentation were successfully minimized. So, it is considered that the proposed algorithms are able to handle the minimization of process invocation graph.

5.4. Processing of Minimized Process Invocation Graph

```
function generateInstructions(seq):
    for el in seq:
        if el in labelMap:
            instructions += labelMap[el]
        if el.hasGuard:
            var = variableMap[el]
            instructions += Check<var, el.guard>
        if el in jumpMap:
            instructions += jumpMap[el]
        if el is ProcessInvocation:
            process = processMapping[el.process]
            params = []
            for df in process.inputs:
                for p in df.parameters:
                    key = (d, p)
                    var = variableMap[key][1]
                    params += var
            resultingValues = []
            for df in process.outputs:
                for p in df.parameters:
                    resultingValues += (df, p.type, p.name)
            resultVar = null
            additionalInstructions = []
            if resultingValues.size == 1:
                key = (resultingValues[0][0],
                        resultingValues[0][1])
                resultVar = variableMap[key][1]
            else if resultingValues.size != 0:
                resultVar = variableMap[el.process][1]
                for rv in resultingValues:
                    key = (rv[0], rv[1])
                    var = variableMap[key][1]
                    attr = (rv[2], rv[1])
                    additionalInstructions
                            += GetField<resultVar, attr, var>
            instructions += Invoke<process, params, resultVar?>
            for ai in additionalInstructions:
                instructions += ai
        if el.hasChildren:
            generateInstructions(el.children)
```

Fig. 5.9. Instruction generation.

As a result of process invocation graph minimization, a single vertex containing all the information on the initial graph is obtained. It is possible to see that this information is already defined in a sequential way – vertex content is created during the merging of other vertices. Also, for each process in a source business process model a method signature exists. At this moment it is still unclear to which class should each method belong, however, method parameters and results are already defined.

After each process invocation it is possible to get one or more results, which, in turn, means that the method corresponding to the appropriate process might return one or more concepts or primitive data types. There are programming languages that allow for multiple

returns – directly (for example, Python [95], Ruby [77]) or by using method/function parameters (for example, C [76], C++ [81], C# [12]). However, other languages would require special approach, such as an additional class introduction to collect the invocation results. So, to achieve maximal universality of the proposed transformation, the author decides to define the so-called result classes, which will contain multiple outputs as attributes.

Minimized process invocation graph can be converted to the instruction sequence by performing the following in the first place:

- define variables to be used during instruction execution;
- define labels to mark branching points;
- define branching possibilities;
- define result classes.

After these steps are performed, it is possible to generate instructions by processing the contents of minimized graph's single vertex. The algorithm responsible for instruction sequence creation is presented in Fig. 5.9.

6. EVALUATION OF CODE GENERATION ALGORITHM

6.1. Validation of Algorithm Execution Results

To make sure that the generated instruction sequence correctly corresponds to the source model, it is necessary to perform algorithm execution results validation. It can be done in multiple ways – for example, one can compare the code produced by the algorithm to the code produced by a human developer. It is also possible to analyse the generated instruction sequence. Other validation techniques might exist, but in this Thesis, the one based on comparison of two graphs is selected. The first graph in this comparison is the process invocation graph that fully corresponds to the initial business process model. The second graph can be created from the generated instruction sequence, and it is called transformation validation graph.

To create a transformation validation graph, it is necessary to analyse the instruction sequence. During this analysis graph vertices that are either process invocation instructions or labels are defined. After vertices are created, it is possible to define validation graph edges, which, in turn, are possible execution flows – if process B is executed after process A, then edge A → B is added to the graph. Branching instructions are processed in a similar way – if branching to the label is possible X, then an appropriate edge is added. As a result, the graph containing information on all possible process invocation sequences is created. It is possible to see that such graph should correspond to the initial process invocation graph. If initial graph contains vertices A and B, and between these vertices only two possible paths – A → C → B and A → D → B exist, then both vertices should be present in a validation graph. Also, both paths should be possible (with no additional ones). If these conditions are not met, transformation is unsuccessful.

When comparing both graphs, it is necessary to remember that the validation graph contains additional vertices that correspond to labels. So, it is not possible to perform such a comparison in a primitive way – by trying to find a corresponding edge in a validation graph for each initial graph's edge. It is necessary to check if the path between appropriate vertices exists. It is also necessary to make sure that this path consists only of both vertices and, possibly, labels. If the found path contains additional process, transformation is unsuccessful.

So, it is necessary to solve two tasks. To check if a path exists between two graph vertices, it is possible to use Floyd–Warshall algorithm [22], Iterative Deepening Depth-First Search (IDDFS) [46] or other algorithms.

When it is confirmed that the path exists between two validation graph vertices, it is necessary to analyse if this path consist only of both processes and, possibly, labels. To solve this task, it is possible to use backtracking [45] approach, which allows to identify if only the allowed vertices exist in the path being analysed.

After solving both tasks – path existence check and its validation – it is possible to define a validation algorithm that compares the process invocation graph to process validation graph in order to verify that all the information that was contained in the initial model was kept after

the transformation had finished. Such an algorithm should analyse the edges of the process invocation graph in the following way:

- if the transformation validation graph does not contain appropriate edge's source or target vertex, transformation is unsuccessful;
- if transformation validation graph does not contain a path between the edge's source and target vertex, transformation is unsuccessful.
- if the aforementioned path contains not only labels, transformation is unsuccessful.

In the case when all the process invocation graph edges were analysed and no aforementioned errors were found the transformation was successful.

6.2. Java Code Generation From Instruction Sequence

After the instruction sequence is generated, it is possible to use it to obtain a software code. In the Doctoral Thesis one of the possible Java [41] code generation approaches is described.

Java programming language is compiled into a so-called bytecode. So, it should be possible to replace each of the generated instructions with one or more Java bytecode elements. This should lead to the same result as the Java code compilation. However, in this case the compiled Java classes are generated instead of code.

However, it is possible to use bytecode to obtain the source code. The Java bytecode is relatively simple in comparison to a typical assembly language. For example, the Java byte code consists of ~200 instructions [16], while the assembly language for Intel CPUs consists of ~2000 instructions [37]. As a result, it is possible to decompile the Java bytecode to convert it to initial source code.

Decompilation is one of the reverse engineering approaches, which is meant to obtain an initial source code from the compiled artifacts [19]. In a broader sense, it is an initial knowledge or initial artifact extraction process from everything manmade. In the beginning reverse engineering and decompilation seems illegal, since it might lead to copyright infringement. However, it is possible to see that many valid examples of legal reverse engineering exist.

- It might be necessary to fix the defects in a software developed some time ago, when initial source code is no longer available.
- Necessity for the reverse engineering can also occur if it is necessary to extract information from the software artifact (cryptographic keys as an example). Of course, to keep the process legal, such artifacts should be self-developed and without available source code.
- Yet another example of a legal reverse engineering is the analysis of computer viruses performed by antiviral software developers to understand how malware works, and what actions should be taken for threat neutralization.

It is possible to find even more examples, however, in this Thesis, decompilation possibilities are utilised to obtain source code from the generated instruction sequence.

To have a decompilable artifact, it is necessary to create a binary Java class that corresponds to the generated instruction sequence. Since the defined instructions were already based on Java bytecode [32], it is possible to define simple rules for its conversion into bytecode elements. Each of the instruction used corresponds to the sequence of Java bytecode elements. This information is presented in Table 6.1. Here, only instructions that are necessary for code generation, are presented.

<div align="right">Table. 6.1</div>

<div align="center">Instruction Mapping to Java Bytecode</div>

Instruction	Java bytecode
`Label<Name>`	In the case of bytecode, offsets from method start are used as labels [15]
`Var<Id, Name, Type>`	Specific instructions do not exist. Variable tables are used instead [15]

`GetField<Object, Field, Target>`	`ALOAD Object`
	`GETFIELD Field`
	`ASTORE Target`

`Invoke<Method, Inputs, Output?>`	`ALOAD 0`
	`∀ v ∈ Inputs: ALOAD v`
	`INVOKESPECIAL Method …`
	If the method returns one or more results:
	`    ASTORE Output`
	In case the object is returned, it is necessary to decompose this object into local variables:
	`    ∀ f ∈ Output.fields:`
	`        ALOAD Output`
	`        GETFIELD f`
	`        ASTORE var`

`Jump<Label>`	`GOTO Label`

`Check<Var, Guard>`	Method invocation preconditions are defined in a free language, so it is necessary to insert an appropriate comment into the code. This comment should help to define appropriate condition later. Java bytecode does not contain comments, so it is proposed to use `valueOf()` method of `Boolean` class to process such cases:
	`    LDC Guard`
	`    INVOKESTATIC Boolean.valueOf`
	`    ISTORE Var`

`JumpIf<Var, Label>`	Two steps are necessary. The first one is variable loading:
	`    ILOAD Var`
	Then, conditional branching is performed:
	`    IFEQ Label`

`JumpIfNot<Var, Label>`	`ILOAD Var` `IFNE Label`

`Return<Var?>`	If `Var` is defined, it is necessary to load it:
	`    ALOAD Var`
	And return it:
	`    ARETURN`
	Otherwise, simple method return is used:
	`    RETURN`

To convert instructions to bytecode elements, ASM [5] library is used.

After the bytecode is generated, it is possible to decompile it obtaining Java source code. To do so, a special software called decompiler is needed. In the case of Java multiple decompilers research [30] was dedicated to the comparison of four such programs – JAD [42], CFR [14], Procyon [52], and FernFlower [26]. As a result, Procyon decompiler was selected to perform initial source code obtaining.

After the decompilation is performed, it is possible to combine its results with process invocation result classes as well as with the classes that have been generated from concepts. As a result, a set of classes is created containing both domain classes as well as the class implementing business logic. It is possible to process this class set with an improved class relationship detection algorithm.

A resulting source code is generated according to anaemic data model defined principles – data (domain objects) is separated from its processing logic [21], [23], [49]. Based on his experience in a software development, the author concludes that such a code is more suitable for the modern enterprise system development. However, anaemic data model is not only a possible variant of the produced code – during the transformations method, signatures and business logic supporting code is generated. So, it is possible to combine this information with already generated domain classes (for example, using the approach from [61] or [66]) to produce a rich data model.

6.3. Related Work

An analysis on related work was also performed during the Thesis development. Researches, that utilize state diagrams as a source model, were reviewed. Such researches were selected due to the fact that the proposed algorithm processes similar data structure, as well as the fact, that UML class, sequence and activity diagrams can be transformed into a source code by using simplified rules, since these diagrams already describe ready solutions. During the Thesis development, researches [7], [53], [83], [91] were analysed. All these works propose methods that are meant to produce a correct executable code. Also, their authors usually set an additional goal to generate minimal amount of source code with maximal performance. This is achieved by using different state machine representations, utilizing multithread data processing, as well as minimizing the size of executable code. While this allows the research authors to complete the defined goal, it also leads to a code that might be hardly readable or maintainable. So, it can be concluded that the methods of analysed researches are weakly suited for the MDSD support.

Also, researches focusing on intermediate models for code generation were analysed. In [48], [72], [94], it is possible to see, that the proposed methods usually define object-oriented languages as a target. The intermediate model proposed in this Thesis, in contrast, is not language paradigm-dependent, which is shown in the next section by generating a procedural code.

6.4. Code Generation Algorithm Evaluation Results

As it is demonstrated, the proposed code generation algorithm allows code generation from an intermediate model. The intermediate model, in turn, is produced from the source model – two-hemisphere model. To validate the correctness of such a transformation, the author of the Thesis also proposed a transformation validation algorithm, which compares the produced intermediate model to a source model.

To generate the source code in selected programming language, decompilation approach was chosen [19]. To achieve this, intermediate model instructions are converted into Java Virtual Machine (JVM) [16] instructions, that are later used as a source model for the code Java code generation. The fact that the produced byte code can be correctly decompiled also serves as proof for the transformation rule correctness.

In the Doctoral Thesis an example that was created to contain multiple cases that require special processing (for example, loops with multiple entry and exit points, branching, data flow without assigned concepts) is used to test the proposed transformation rules. As a result, a source two-hemisphere model is converted into a correct Java code that can be further modified.

In this section a related work is also described noting main limitations of the proposed methods – its authors mainly use UML [90] as a source model and object-oriented languages as a target. The studies proposing methods for a code generated from state diagrams also use different state machine representations and optimizations producing code, that is hard to read and maintain. No attempts to generate a different kind of source code were proposed.

7. PRACTICAL APPLICATION OF THE THESIS RESULTS

During the development of the Doctoral Thesis its author was offered to develop a program for a PIC18F45Q10 [73] microcontroller. Together with the customer it was decided to apply the approach described in the Thesis and define a two-hemisphere model for the developed system. Later that model was used to generate the system source code. Electric guitar effects are devices used to modify the sound of electric guitar [36], for example, add echo or modify the audio signal in other way.
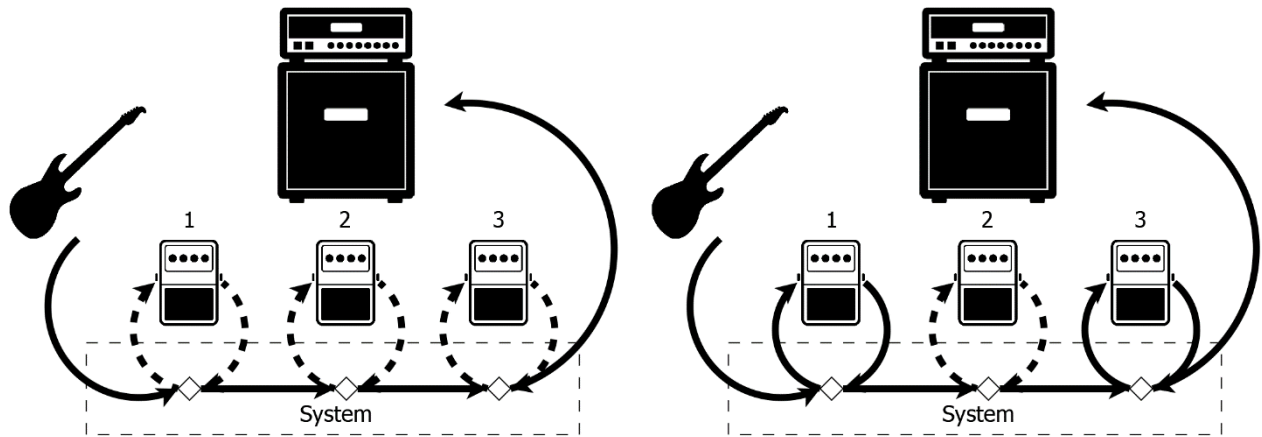

Fig. 7.1. Electric guitar effect switching system.

The switching system allows for a so-called effect loop creation. These loops can be used for multiple effect commutation by including them into the signal path or excluding from it. The signal path begins with an electric guitar and ends with an amplifier. It is possible to have multiple effects in it. Figure 7.1 shows two different situations. On the left side of the figure, all the effects are excluded, while on the right, effects 1 and 3 are processing the signal. Of course, it is possible to define other signal paths as well. Microcontroller is responsible for effect commutation and allows to create presets with a defined signal path. These presets can be saved and reused.

Microcontroller software is defined using C programming language [40], [76], so it was necessary to modify the code generation algorithm. However, the amount of necessary modifications was minimal, which proves the fact that the proposed method is not limited with object-oriented code generation.

So, the proposed method was also used in practice, and its author's experience during interoperation with the customer proves that it can also be used in a larger project development.

CONCLUSIONS AND RESULTS

In the Doctoral Thesis code generation from the source model in context of MDSD is described. Code generation itself was widely used from 1980s to serve different purposes (for example, in integrated development environments – IDEs), however, in model driven software development it is not only a supporting tool but also a core technique. If it is possible to produce a code from the source model that is both understandable to software developers and problem area experts, then it is possible to consider a full MDSD support. In the Thesis, notations of the different source models are analysed and the appropriate one is selected. The selected model can already be used in MDSD context on different levels of abstraction, and in this work, it is modified to support the code generation task to support the model-driven approach on other not yet covered levels. During the development of the Thesis, all tasks were successfully accomplished.

1. An analysis of two-hemisphere model advantages and limitations in code generation context was performed. Several limitations to be fixed in this context were identified. For all the identified limitations solutions were offered thus supporting further work.

2. A target programming language was selected based on but not limited to the TIOBE programming language popularity rating. As a result, the Java programming language was chosen.

3. Transformation rules allowing code generation from the two-hemisphere model were developed. These rules use an intermediate model for code generation, so it is possible to support not only the object-oriented paradigm, which was also proven practically.

4. A validation methodology to verify the correctness of transformations was developed. This methodology is based on the source model and transformation result comparison.

5. By utilizing the proposed approach, a software for an electric guitar effect switching system was developed. This allowed to approbate the method practically as well.

The main result of the Doctoral Thesis is the proposed algorithm that is used to generate a source code from the two-hemisphere model. In the Thesis, Java code generation is described, however, the algorithm itself is defined in a general way and allows for the code generation in different programming languages that are not limited to the object-oriented ones. To achieve this, an intermediate model based on a special instruction set is used.

An additional result, which was not a part of the initial task definition, was achieved. However, without it the definition of the algorithm would not be possible. This result is a modernization of the two-hemisphere model. To solve the limitations in the current notation, additional elements to be added to it are proposed.

In general, it can be considered that **the results of the Doctoral Thesis** are as follows.

1. A two-hemisphere model was improved and enriched with additional elements.
2. Two-hemisphere model transformation rules in a form of pseudocode were defined.
3. An intermediate model for code generation support was developed. This model can also be used along with other MDSD methods.

4. An improved class relationship detection algorithm was defined. This algorithm can also serve for different purposes outside of the research scope. For example, it should be possible to use it for code analysis and refactoring.

5. The proposed method was also approbated in the practical task solving for a software system development.

Based on the research conducted during the Thesis development and the achieved results it is possible to **conclude the following**.

1. While in the context of MDSD the UML language is used to define the source model, the author of the Thesis based on the performed analysis and survey considers that it is not well suited for an initial input to the code generation algorithm. This can be explained by the fact that UML is meant to describe existing solutions and can be poorly understood by the business area experts.

2. The two-hemisphere model can be read and understood by both the software developers as well as other stakeholders (it was proven during practical application of the proposed method – the customer himself could define the model with minimal assistance). It means that such a model can be used as a source model in the MDSD area.

3. The two-hemisphere model can be used not only to produce UML diagrams but also the software code.

4. Additional results, i.e., a class relationship detection algorithm and intermediate model for code generation, were also produced during the Thesis development. It is possible to use these not only for the two-hemisphere model transformation but also in conjunction with other source models as well. It is also possible to note that an intermediate model allows for a non-object-oriented code as well.

5. Based on the research performed during the Thesis development, the proposed improvements both to the model notation and its transformation rules, it is possible to further enrich two-hemisphere model transformation algorithms or to develop a tool for its support.

So, by modifying the existing two-hemisphere model notation and adapting it to support the code generation it was possible to define the transformation rules that can be now enclosed into a supporting tool to allow wider adoption in the enterprise. The proposed transformation rules themselves allow for a software code generation from a model that can be read and understood by problem area experts. The results of this research can be useful for a wide range of specialists both in enterprise, as well as in MDSD research area.

BIBLIOGRAPHY

1. Águila, I., Palma, J., Túnez, S. Milestones in Software Engineering and Knowledge Engineering History: A Comparative Review. *The Scientific World Journal*, 2014, vol. 14, 10 pages. Available: doi:10.1155/2014/692510

2. Anderson, J. R. *Cognitive psychology and its implications*. 7th edition. USA: Worth Publishers, 2009. 469 p. ISBN 978-1429219488.

3. *AngularJS: Developer Guide: Conceptual Overview*. [Accessed 9 December 2019]. Available : https://docs.angularjs.org/guide/concepts

4. *ANTLR*. [Accessed 9 December 2019]. Available: http://www.antlr.org/

5. *ASM*. [Accessed 9 December 2019]. Available: https://asm.ow2.io/

6. Asnina, E. The Formal Approach to Problem Domain Modeling Within Model Driven Architecture. In: *Proceedings of the 9th International Conference "Information Systems Implementation and Modeling" (ISIM 2006)*. Ostrava, Czech Republic, 2006, pp. 97-104.

7. Badreddin, O., Lethbridge, T., Forward, A., Elaasar, M., Aljamaan, H. Garzón, M. Enhanced Code Generation from UML Composite State Machines. In: *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*. Lisbon, Portugal, 2014, pp. 235–245. ISBN 978-989-758-007-9. Available from: doi:10.5220/0004699602350245.

8. Baudoin, C. R. The Evolution and Ecosystem of the Unified Modeling Language. In: *Proceedings of Present and Ulterior Software Engineering (PAUSE) symposium*. USA: Springer International Publishing AG, 2017, pp. 37–46. ISBN 978-3-319-67424-7. Available from: doi:10.1007/978-3-319-67425-4_3.

9. Benoît, L., Jitia, C.-E., Jouenne, E. DSL classification. In: *Proceedings of OOPSLA 7th workshop on domain specific modeling*. USA: ACM, 2007.

10. *BPMN Specification – Business Process Model and Notation*. [Accessed 2019.g. 9.decembrī]. Pieejams: http://www.bpmn.org/

11. Brambilla, M., Cabot, J., Wimmer, M. *Model-Driven Software Engineering in Practice: Second Edition. 2nd edition*. USA: Morgan & Claypool Publishers, 2017. 208 p. ISBN 978-1627057080.

12. *C# Programming Guide*. [Accessed 12 December 2019]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/

13. Cemus, K., Cerny, T., Matl, L., Donahoo, M. J. Aspect, Rich and Anemic Domain Models in Enterprise Information Systems. In: *Proceedings of the 42nd International Conference on SOFSEM 2016: Theory and Practice of Computer Science*. Berlin, Germany: Springer-Verlag, 2016, pp. 445–456. ISBN 978-3-662-49191-1. Available from: doi:10.1007/978-3-662-49192-8_36.

14. *CFR – yet another Java decompiler*. [Accessed 12 December 2019]. Pieejams: http://www.benf.org/other/cfr/

15. *Chapter 4. The class File Format*. [Accessed 12 December 2019]. Available: https://docs.oracle.com/javase/specs/jvms/se13/html/jvms-4.html

16. *Chapter 6. The Java Virtual Machine Instruction Set*. [Accessed 12 December 2019]. Pieejams: https://docs.oracle.com/javase/specs/jvms/se13/html/jvms-6.html

17. Chen, P.P. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS) - Special issue: papers from the international conference on very large data bases: September 22–24*. 1976, Volume 1, Issue 1, pp. 9–36. Available from: doi:10.1145/320434.320440.

18. Daniel, F., Matera, M. *Mashups: Concepts, Models and Architectures*. Berlin, Germany: Springer-Verlag, 2014. 319 p. ISBN 978-3-642-55048-5.

19. Eilam, E. *Reversing: Secrets of Reverse Engineering*. 1st edition. USA: Wiley, 2005, 624 p. ISBN 978-0764574818.

20. Epp, S. S. *Discrete Mathematics with Applications*. 4th Edition. USA: Cengage Learning, 2010. 984 p. ISBN 978-0495391326.

21. Evans, E. *Domain-driven design: tackling complexity in the heart of software*. USA: Addison-Wesley Professional, 2003. 560 p. ISBN 978-0321125217.

22. Floyd, R. W. Algorithm 97: Shortest path. *Communications of the ACM Magazine*, 1962, Volume 5, Issue 6, p. 345. Available from: doi:10.1145/367766.368168.

23. Fowler, M. *Anaemic Domain Model*. [Accessed 13 December 2019]. Available: http://www.martinfowler.com/bliki/AnemicDomainModel.html

24. Fowler, M. *Refactoring: Improving the Design of Existing Code*. 2nd Edition. USA: Addison-Wesley Professional, 2018. 486 p. ISBN 978-0201485677.

25. Gane, C., Sarson, T. *Structured systems analysis: tools and techniques*. USA: Prentice-Hall, 1979. 241 p. ISBN 978-0138545475.

26. *GitHub - fesh0r/fernflower: Unofficial mirror of FernFlower Java decompiler*. [Accessed 13 December 2019]. Available https://github.com/fesh0r/fernflower

27. Grundspenkis, J. Causal Domain Model Driven Knowledge Acquisition for Expert Diagnosis. *System Development. Journal of Intelligent Manufacturing*, 1998, Volume 9, Issue 6, pp. 547–558. eISSN 1572-8145. ISSN 0956-5515. Available from: doi:10.1023/A:1008840303610.

28. Grune, D., Jacobs, C. J. H. *Parsing Techniques: A Practical Guide (Monographs in Computer Science)*. 2nd Edition. USA: Springer, 2010. 688 p. ISBN 978-1441919014.

29. Gurad, H. D., Mahalle, V. S. An Approach to Code Generation from UML Diagrams. *IJESRT - International Journal of Engineering Sciences & Research Technology*, 2014. pp. 421–423. ISSN 2277-9655.

30. Gusarovs, K. An Analysis on Java Programming Language Decompiler Capabilities. *Applied Computer Systems*, 2018, Volume 23, No. 2. pp. 109–117. eISSN 2255-8691. ISSN 2255-8683. Available from: doi:10.2478/acss-2018-0014.

31. Gusarovs, K., Nikiforova, O., Jukss, M. A Prototype of Description Language for the Two-Hemisphere Model. *Applied Computer Systems*, 2015, Volume 18, Issue 1. pp. 15–20. ISSN 2255-8691. Available from: doi:10.1515/acss-2015-0014.

32. Gusarovs, K., Nikiforova, O. An Intermediate Model for the Code Generation from the Two-Hemisphere Model. In: *Proceedings of the 14th International Conference on Software Engineering Advances (ICSEA 2019)*, accepted for publishing. ISBN: 978-1-61208-752-8.

33. Gusarovs, K., Nikiforova, O., Giurca, A. Simplified Lisp Code Generation from the Two-hemisphere Model. *Procedia Computer Science*, 2017, Volume 104, Issue C. pp. 329–337. Available from: doi:10.1016/j.procs.2017.01.142.

34. Gusarovs, K., Nikiforova, O. Workflow Generation from the Two-Hemisphere Model. *Applied Computer Systems*, 2017, Volume 22, Issue 1. pp. 36–46. eISSN 2255-8691. ISSN 2255-8683. Available from: doi:10.1515/acss-2017-0016.

35. *Hibernate. Everything data. – Hibernate*. [Accessed 13 December 2019]. Available: http://hibernate.org/

36. Hunter, D. *Guitar Effects Pedals - the Practical Handbook*. Canada: Backbeat Books, 2004. 192 p. ISBN 978-0879308063.

37. *Intel® 64 and IA-32 Architectures Software Developer Manuals | Intel® Software*. [Accessed 13 December 2019]. Available https://software.intel.com/en-us/articles/intel-sdm

38. *ISO/IEC 2382:2015(en), Information technology — Vocabulary*. [Accessed 13 December 2019]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en

39. *ISO - ISO/IEC 9075-1:2016 - Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. [Accessed 13 December 2019]. Available https://www.iso.org/standard/63555.html

40. *ISO – ISO/IEC 9899:2011 – Information technology – Programming languages – C*. [Accessed 13 December 2019]. Available: https://www.iso.org/standard/57853.html

41. *Java | Oracle*. [Accessed 13 December 2019]. Available https://www.java.com/en/

42. *Java Decompiler*. [Accessed 13 December 2019]. Available: http://java-decompiler.github.io/

43. Johnson, D. B. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 1975, Volume 4, pp. 77-84. Available from: doi: 10.1137/0204007.

44. Kleppe, A., Warmer, J., Bast W. *MDA Explained: The Model Driven Architecture – Practise and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. 170 p. ISBN 032119442X.

45. Knuth, D. E. *The Art of Computer Programming*. 1st Edition. USA: Addison-Wesley Professional, 2003. 9998 p. ISBN 978-0321751041.

46. Korf, R. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence Journal*, 1985, Volume 27, Issue 1, pp. 97–109. ISSN 0004-3702. Available from: doi:10.1016/0004-3702(85)90084-0.

47. Koshy, T. *Discrete Mathematics With Applications*. 1st Edition. USA: Academic Press, 2003. 1042 p. ISBN 978-0124211803.

48. Lasbahani, A., Chhiba, M., Tabyaoui, A. A UML Profile for Security and Code Generation. *International Journal of Electrical and Computer Engineering (IJECE)*, 2018, Volume 8, No. 6, pp. 5278–5291. ISSN 2088-8708. Available from: doi:10.11591/ijece.v8i6.pp5278-5291.

49. Luís Marques. *A defense of so-called anemic domain models. Slides of D-Lang-Silicon-Valley Meetup @ January 28, 2016*. [Accessed 24 December 2019]. Available http://files.meetup.com/18234529/luis_marques_anemic_domain_models.pdf

50. Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st Edition. USA: Prentice Hall, 2008. 464 p. ISBN 978-0132350884.

51. Mernik, M., Heering, J., Sloane, A.M. When and how to develop domain-specific languages. *ACM Computing Surveys*, 2005, Volume 37, No. 4, pp. 316–344. ISSN 0360-0300. Available from: doi:10.1145/1118890.1118892.

52. *mstrobel / procyon – Bitbucket*. [Accessed 24 December 2019]. Available: https://bitbucket.org/mstrobel/procyon

53. Niaz, I. A., Jiro, T. Mapping UML statecharts to Java code. In: *IASTED Conf. on Software Engineering*, 2004.

54. Nikiforova, O. System Modeling in UML with Two-Hemisphere Model Driven Approach. *Applied Computer Systems*, 2011, Volume 41, Issue 1, pp. 37–44. ISSN 2255-8691. Available from: doi:10.2478/v10143-010-0022-x.

55. Nikiforova, O. Two Hemisphere Model Driven Approach for Generation of UML Class Diagram in the Context of MDA. *e-Informatica Software Engineering Journal*, 2009, Volume 3, pp. 59–72.

56. Nikiforova, O., Bohomaz, Y., Gusarovs, K. A Comparison of the Implementation Means for Development of Modelling Tool. In: *Proceedings of the 2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS 2017)*, 2017, pp. 1–6. Available from: doi:10.1109/WITS.2017.7934623.

57. Nikiforova, O., El Marzouki, N., Gusarovs, K., Vangheluwe, H., Bures, T., Al-Ali, R., Iacono, M., Esquivel, P. O., Leon, F. The Two-Hemisphere Modelling Approach to the Composition of Cyber-Physical Systems. In: *In Proceedings of the 12th International Conference on Software Technologies*. Portugal: SciTePress, 2017, pp. 286–293. ISBN 978-989-758-262-2. Available from: doi:10.5220/0006424902860293.

58. Nikiforova, O., Gorbiks, O., Gusarovs, K., Ahilcenoka, D., Bajovs, A., Kozacenko, L., Skindere, N., Ungurs, D. Development of BrainTool for Generation of UML Diagrams from the Two-hemisphere Model Based on the Two-Hemisphere Model Transformation Itself. In: *Proceedings of the International Scientific Conference "Applied Information and Communication Technologies"*. Latvia: LLU, 2013, pp. 267–274. ISSN 2255-8586.

59. Nikiforova, O., Gusarovs, K. Comparison of BrainTool to Other UML Modeling and Model Transformation Tools. *AIP Conference Proceedings*, 2017, Volume 1863, No. 1, id. 330005. Available from: doi:10.1063/1.4992503.

60. Nikiforova, O., Gusarovs, K. Anemic Domain Model vs Rich Domain Model to Improve the Two-Hemisphere Model-Driven Approach. *Applied Computer Systems*, 2020, Volume 25, Issue 1. (Accepted for publication).

61. Nikiforova, O., Gusarovs, K., Gorbiks, O., Pavlova, N. BrainTool. A Tool for Generation of the UML Class Diagrams. In: *Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 2012. Lisbon: IARIA, 2012, pp. 60–69. ISBN 9781612082301.

62. Nikiforova, O., Gusarovs, K., Gorbiks, O., Pavlova, N. Improvement of the Two-Hemisphere Model-Driven Approach for Generation of the UML Class Diagram. *Applied Computer Systems*, 2013, Volume 14, Issue 1, pp. 19–30. ISSN 2255-8691. Available from: doi:10.2478/acss-2013-0003.

63. Nikiforova, O., Gusarovs, K., Ressin, A. An Approach to Generation of the UML Sequence Diagram from the Two-Hemisphere Model. In: *Proceedings of the 11th International Conference on Software Engineering Advances (ICSEA 2016)*. Wilmington: IARIA, 2016, pp. 142–49. ISBN 978-1-61208-498-5.

64. Nikiforova, O., Kirikova, M. Two-Hemisphere Model Driven Approach: Engineering Based Software Development. In: *Advanced Information Systems Engineering. CAiSE 2004. Lecture Notes in Computer Science*, Volume 3084. Berlin: Springer, 2004, pp. 219–233. ISBN 978-3-540-22151-7. Available from: doi:10.1007/978-3-540-25975-6_17.

65. Nikiforova, O., Kozacenko, L., Ahilcenoka, D. UML Sequence Diagram: Transformation from the Two-Hemisphere Model and Layout. *Applied Computer Systems*, 2013, Volume 14, Issue 1, pp. 31–41. ISSN 2255-8691. Available from: doi:10.2478/acss-2013-0004.

66. Nikiforova, O., Kozacenko, L., Ungurs, D., Ahilcenoka, D., Bajovs, A., Skindere, N., Gusarovs, K., Jukss, M. BrainTool v2.0 for Software Modeling in UML. *Applied Computer Systems*, 2015, Volume 16, Issue 1, pp. 33–42. ISSN 2255-8691. Available from: doi:10.1515/acss-2014-0011.

67. Nikiforova, O., Kozacenko, L., Ahilcenoka, D., Gusarovs, K., Ungurs, D., Jukss, M. Comparison of the Two-Hemisphere Model-Driven Approach to Other Methods for Model-Driven Software Development. *Applied Computer Systems*, 2016, Volume 18, Issue 1, pp. 5–14. ISSN 2255-8691. Available from: doi:10.1515/acss-2015-0013.

68. Nikiforova, O., Pavlova, N. Development of the Tool for Generation of UML Class Diagram from Two-hemisphere model. In: *Proceedings of The Third International Conference on Software Engineering Advances (ICSEA)*. USA: IEEE, 2008, pp. 105–112. ISBN 978-1-4244-3218-9.

69. Nikiforova, O., Pavlova, N., Gusarovs, K., Gorbiks, O., Vorotilovs, J., Zaharovs, A., Umanovskis, D., Sejans, J. Development of the Tool for Transformation of the Two-Hemisphere Model to the UML Class Diagram: Technical Solutions and Lessons Learned. In: *Proceedings of the 5th Internation Scientific Conference "Applied Information and Communication Technology 2012"*. Latvia: LLU, 2012, pp. 11–19. ISBN 978-9984-48-065-7.

70. Nikiforova, O., Sukovskis, U., Gusarovs, K. Application of the Two-Hemisphere Model Supported by BrainTool: Football Game Simulation. *AIP Conference Proceedings*, 2015, Volume 1648, No. 1, id. 310004. Available from: doi:10.1063/1.4912557.

71. Noureen, A., Amjad, A., Azam, F. Model Driven Architecture - Issues, Challenges and Future Directions. *JSW*, 2016, Volume 11, pp. 924–933. Available from: 10.17706/jsw.11.9.924-933.

72. Omar, E. B., Brahim, B., Taoufiq, G. Automatic code generation by model transformation from sequence diagram of system's internal behavior. *International Journal of Computer and Information Technology*, 2012, Volume 1, Issue 2, pp. 129–146.

73. *Overview: PIC18F45Q10 - 8-bit Microcontrollers.* [Accessed 25 March 2020]. Available https://www.microchip.com/wwwproducts/en/PIC18F45Q10

74. Paige, R. F., Zolotas, A., Kolovos, D. The Changing Face of Model-Driven Engineering. In: *Proceedings of Present and Ulterior Software Engineering (PAUSE) symposium*. Germany: Springer, 2017, pp. 103–118. ISBN 978-3-319-67424-7.

75. Perisic, B. Model Driven Software Development – State of the Art and Perspectives. In: *Proceedings of INFOTEH 2014*, Volume 13. pp. 1237–1248.

76. Ritchie, D. M., Kernighan, B. W. *The C Programming Language*. Second Edition. USA: Prentice Hall, 1988. 272 p. ISBN 978-0131103627.

77. *Ruby Programming Language*. [Accessed 26 December 2019]. Available: https://www.ruby-lang.org/en/

78. Salomon, D. *Assemblers and Loaders*. USA: Prentice Hall, 1993. 308 p. ISBN 978-0130525642.

79. Shiferaw, M. K., Jena, A. K. Code Generator for Model-Driven Software Development Using UML Models. *Proceedings of 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2018, pp. 1671–1678. Available from: doi:10.1109/iceca.2018.8474690.

80. Skiena, S. S. *The Algorithm Design Manual*. 2nd Edition. Germany: Springer, 2008. 748 p. ISBN 978-1849967204.

81. *Standard C++*. [Accessed 26 December 2019]. Available https://isocpp.org/

82. Stevens, W. P., Myers, G. J., Constantine, L.L. Structured Design. *IBM Systems Journal*, 1974, Volume 13, Issue 2, pp. 115–139. Available from: doi:10.1147/sj.132.0115.

83. Sunitha, E. V., Philip, S. Automatic Code Generation From UML State Chart Diagrams. *IEEE Access*, 2019, Volume 7, pp. 8591–8606. ISSN 2169-3536. Available from: doi:10.1109/ACCESS.2018.2890791.

84. Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1972, Volume 1, Issue 2, pp. 146–160. Available from: doi:10.1137/0201010.

85. Terekhov, A., Bryksin, T., Litvinov, Y. How to Make Visual Modeling More Attractive to Software Developers. In: *Present and Ulterior Software Engineering*. Germany: Springer, 2017, pp. 139–152. ISBN 978-3-319-67424-7.

86. *The Anaemic Domain Model is no Anti-Pattern, it's a SOLID design | SAPM: Course Blog*. [Accessed 2019.g. 26.decembrī]. Available: https://blog.inf.ed.ac.uk/sapm/2014/02/04/the-anaemic-domain-model-is-no-anti-pattern-its-a-solid-design/

87. *The LEX & YACC Page*. [Accessed 26 December 2019]. Available http://dinosaur.compilertools.net/

88. *TIOBE Index | TIOBE – The Software Quality Company*. [Accessed 1 December 2018]. Available: https://www.tiobe.com/tiobe-index/

89. *Trygve/MVC*. [Accessed 26 December 2019]. Available http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

90. *Unified Modeling Language*. [Accessed 26 December 2019]. Available: https://www.omg.org/spec/UML/

91. Van Cam, P., Radermacher, A., Gérard, S., Shuai, L. Complete Code Generation from UML State Machine. *MODELSWARD 2017*, 2017, pp. 208–219. Available from: doi:10.5220/0006274502080219.

92. van der Aalst, W.P.M. Business process management: A comprehensive survey. *ISRN Software Engineering, 2013*, Volume 2013, id. 507984, pp. 1–37. Available from: doi:10.1155/2013/507984.

93. *W3C XML Schema*. [Accessed 26 December 2019]. Available http://www.w3.org/XML/Schema

94. Wang, Z. A JAVA Code Generation Method based on XUML. *IOP Conference Series: Materials Science and Engineering*, 2019, Volume 563, id. 052001. Available from: doi:10.1088/1757-899x/563/5/052001.

95. *Welcome to Python.org*. [Accessed 26 December 2019]. Available: https://www.python.org/

96. Wright, D. R., *Finite State Machines (CSC215 Class Notes)*. [Accessed 20 September 2017]. Available: http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf