Verification Automation of UML Diagrams Created by Students

Tatiana Gasheva
Mathematics and Mechanics Faculty
Perm State University
Perm, Russia
gasheva_99@mail.ru

Andrey Otinov
Mathematics and Mechanics Faculty
Perm State University
Perm, Russia
otinovandry@gmail.com

Dmitry Vlasov
Mathematics and Mechanics Faculty
Perm State University
Perm, Russia
dima.vlasov@icloud.com

Natalia Datsun
Computer Science Department
Mathematics and Mechanics Faculty
Perm State University
Perm, Russia
nndatsun@inbox.ru

***Abstract*— Unified Modeling Language (UML) is the current notation standard (ISO/IEC 19505:2012) to visualize models in software development. UML provides essential guidelines and rules to visualize and understand complex software systems. This is the reason why it has become part of curricula for software engineering courses at many universities worldwide.**

However, many students, teachers or software developers make mistakes when constructing or miss these on checking the correctness of these diagrams. This paper presents software that can help to solve this problem.

Keywords— verification, UCD, AD, CD

I. Introduction

UML is a standard that provides system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes [1]. This standard is widely used in the software industry. Today, the systems are becoming more and more complex and finding errors in models at an early stage can reduce the time and material costs for development.

On the one hand, it is used in Object-oriented analysis and design (OOAD) in the development of complex systems [2]. Formal methods are used to validate such models [3], [4].

On the other hand, a preparation of IT professionals involves the learning modeling process [5] and Model Driven Architecture® (MDA®) [6]. Studies of student perception of UML modeling indicate that this process is perceived as quite complex. This opinion is shared by both computer scientists [7], [8] and computer science majors [7].

The process of manual verification and validation can take a considerable amount of time. This is especially evident during the review of dozens of student models by the teacher. Therefore, the creation of such a system is actual.

Methods for verification use case diagrams (UCD), activity diagrams (AD) and class diagrams (CD) are presented in this article. Verification is defined as "the process of determining that a model or simulation implementation accurately represents the developer's conceptual description and specification" [9]. This paper focuses on verification each type of diagram separately, without maintaining consistency between different UML models. However, all these verification methods are combined in one system, which allows to check any of the described types of diagrams, including in a package mode.

The paper is organized as follows. Section 2 surveys domain knowledge, existing software for drawing and verification of UML diagrams. Section 3 describes our approach to check use case, activity and class diagrams. Section 4 shows the process of realization of the developing system and section 5 and 6 conclude the paper.

II. Analysis

A. Domain Knowledge Analysis

An analysis of the subject area was carried out, which identified and confirmed the need and importance of developing this system for the following groups of users: students and teachers [10], IT industry specialists [11].

B. Analysis Mistakes in Creating UML Diagram Made by Users

Modern approaches to verification student models are based on the use of catalogs or lists of common mistakes [10], [11], [12], [13]. An analysis of existing catalogues was carried out, as well as an own review of this problem based on the works provided by students of Perm State University. The result of this study was a list of lexical, syntactic and semantic mistakes.

C. Analysis Mistakes in Creating UML Diagram Made by Students

An analysis was carried out of 206 the work of IT students who create models of information systems based on an object-oriented approach to modeling.

D. Analysis of Existing Software for Verification of UML Diagrams

For UC verification, two open source software was discovered: *FOAM*[1] tool and *Rational Rose*[2].

For AD verification there are analyzed tools such as *UML-VT*[3], *Woflan*[4].

[1] https://openfoam.org/
[2] www.ibm.com/software/developer/rosexde/
[3] http://www.cs.umd.edu/~rance/projects/uml-vt/
[4] https://www.win.tue.nl/woflan/doku.php

In existing publications [14], there are only brief descriptions of algorithms for verifying CD, and it is impossible to study and analyze them in detail, since they are in the private domain. The set of libraries used in private verification systems: Eclipse (2000 LoC) [15], Java classes (11500 LoC) [16], Dresden OCL toolkit extensible libraries (for processing and loading constraints) [17] and MDR (for importing/exporting UML models from XMI).

These tools did not meet most of the requirements of the target group of users. This confirmed the actuality of the development.

E. *Analysis of UML Diagram Creation Software Providing Metadata*

The choice of the software that will be used for the construction of diagrams is an important step in this work, since the chosen software tool will determine the possibility and success of further analysis, design and implementation of the prototype. That is why special requirements were defined for the selection process of competing modeling systems. The result of the research in this issue was the choice of the *GenMyModel*[5]. It combines a simple user interface, does not require installation and has the function to export the diagram in the required formats. The advantage of this tool is that when building diagrams, it does not allow you to perform some actions that can lead to mistakes. Thanks to this, the list of conditions to be checked can be reduced.

Based on this, it was decided that the input data (XMI and PNG files) will be generated using the GenMyModel tool.

F. *Analysis of UML Diagrams Metadata Exported from GenMyModel*

The data is received in the XML Metadata Interchange (XMI) format [18]. The analysis of the input metadata was carried out and, on the basis of the results obtained, the stages of reading the elements of the diagrams, their unification, storage and processing in the system was designed and implemented.

III. METHODS

A. *Choosing Types of UML Diagrams for Verification Automatisation*

Not all types of UML diagrams were opted for research. To automate the verification, two analysis phase models were chosen - UCD and AD, and one design phase model - CD. This choice is based on the experience of verifying these models and the preferences of the authors of this articles [7], [10].

B. *An Approach to Classifying Student Mistakes*

We propose to classify three groups of mistakes:

1) Model mistakes: Lexical, sintactic, semantic.

2) Positioning mistakes: Visual presentation and positioning of diagram's elements.

Also we will note that the input model does not allow the full determination of positioning mistakes.

C. *Approach for Use-Case Diagram Verification*

A research was carried out among the existing UCD verification methods. For the use case diagrams, the following were identified: Object-Oriented Reading Techniques (OORT) [19] and Checklist-Based Reading (CBR) [20] - reading based on a list of requirements.

Since we have a list of errors, the CBR methodology was chosen. CBR is a very common method. List of mistakes should be checked during the verification. CBR provides more aid and advice to the inspectors than ad-hoc reading and is therefore a very common technique.

List of mistakes, that can be detected are presented.

1) Lexical: System package is missing. Invalid actor name: should be represented by a noun, starting with a capital letter. Incorrect use case name: should be presented as an action (verb or verbal noun) with a capital letter. Missing system name (subject name). Using elements not standarted by UML for UCD. Use cases should be represented by an appropriate element.

2) Syntactic: No extension point for use case with extension relation. Missing text at use-case extension point. Missing text in extension condition. Actor names should be unique. Use case names should be unique. Using the inclusion relationship among actors. Using extension relationships between actors. Finding an actor inside the system. Using an association relationship between use cases. A use case has no links to other elements in the diagram. A use case has no relationship with other diagram elements. A use case should have an association with an actor, or have an extension, addition, or inclusion relationship with other use cases. Using a generalization relationship not between two actors or use cases. Using an include relationship not between two actors or use cases. Using an extension relationship not between two actors or use cases. The actor does not have any association with use cases.

3) Semantic: A parent use case that has an include or extension relationship does not have any association relationship. A use case with an inclusion relation includes only one use case. Two or more actors with the same set of associated use cases. The relationship of inclusion and extension is directed in the wrong direction. Inclusion relation abuse. Non-hierarchical structure of use cases.

Fig.1 is presented possible UCD students mistakes.

Fig. 1. Example of mistakes in UCD.

[5] https://www.genmymodel.com/

D. Approach for Activity Diagram Verification

To solve the AD verification problem, the analysis of existing methods was carried out, such as the construction of a Petri net [21], the use of temporal logic [22], as well as graph transformation systems [23].

For this work we use a subset of UML 2.0 activities. We consider the initial node, final node, decision node, merge node, action node, fork node, join node, swimlane, comment node, flow.

Now we describe the basic idea of the verification process. For each node in AD, there is a class in our system. These classes have three parts: class name, attributes, functions. Class name is a name of node and one of the attributes for node is token attribute [23] and one attribute is a list of links on the next objects. For each AD's node, the object is created and placed in a graph. The graph's vertices represent AD's nodes and the graph's edges represent AD's flows.

The verification is divided into two steps. At the first step lexical, semantic and partial syntax are checked. Then we complete checking syntax by modeling token flow through the graph.

After the model passes the first steps of verification, in order to continue verification, we impose some restrictions. The restrictions are as follows.

1) AD must have exactly one initial node, one or more final nodes and at least one action node.
2) Initial node has no incoming edge and the final node has no outgoing flow.
3) Action, merge, join and init nodes must have exactly one outgoing flow.
4) Fork and decision nodes can have any number of outgoing flows.
5) Action, fork, decision, final nodes should have only one incoming flow.
6) Each join and merge node can have any number of incoming flows.
7) Flows can not start and finish in the same node.

If the restriction was violated, we finish verification without graph checking.

The tokens flow through the graph along the edge directions from initial to final node. The verification is completed when all token flows are checked or a mistake is found. In this case mistakes are the situations when several tokens appear in one node at once or when a token remains in the graph upon reaching the final node or when deadlock occurs (the situation when there is no token can be moved).

According to [23], the graph uses token-flow semantics. The rules are as follows.

1) At the beginning only the init node has a token. Action consumes one input token and creates one token in the output place.
2) Final node consumes one input token.
3) Decision consumes one input token and produces one in any output place.
4) Merge consumes one token from any input place and produces one token in output.
5) Join consumes one token from each input place and creates one token in output.
6) Fork consumes one input token and creates tokens in each output place.

These rules ensure the token flows through the graph.

Fig. 2 presents the original diagram and the graph.

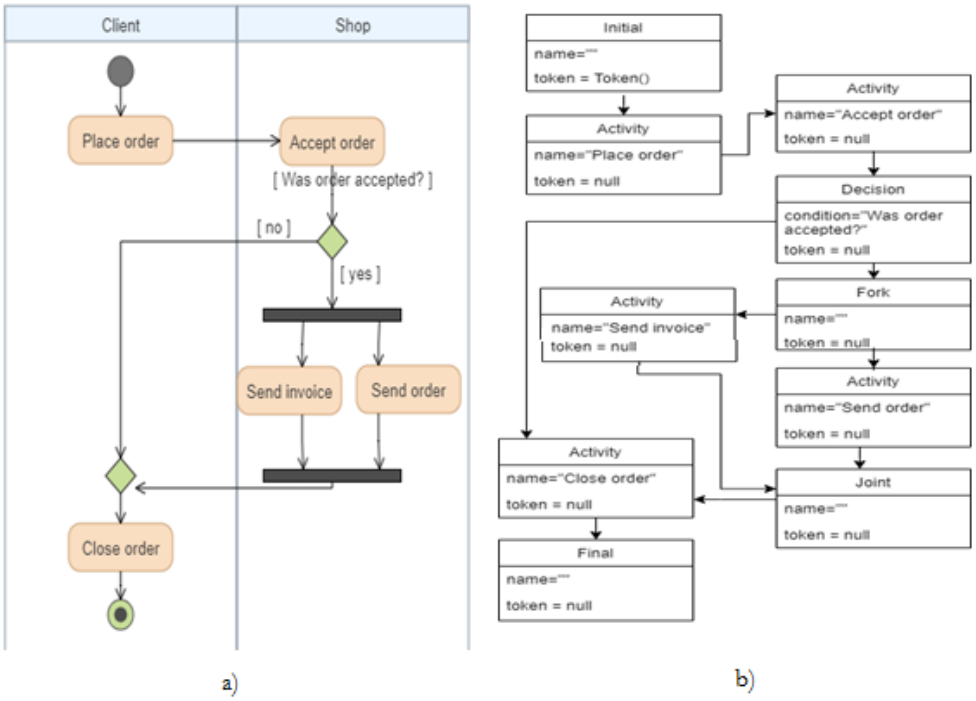

Fig. 2. (a) Sample of AD. (b) Graph represented AD.

The state of the graph at each step can be encoded with a sequence of zeros and ones, where zero means that the element is inactive (does not have a token), and one means that it is active. A stack of current masks and a set of checked masks are created. At each step, the top mask is taken from the current masks and processed. The processing's result is new masks, that are checked for use early using a set of used masks and, if they have not been previously used, are added to the list of current masks.

At each step, all existing tokens are moved to one of the next nodes. In the case of a decision node, a token can activate a random element. So it generates several possible next states that are pushed into the current mask stack.

However, the problem of unpaired use of a joint and a fork remains. Indeed, when there are several fork nodes corresponding to one join nodes, the join can be activated. To figure out this kind of mistake, it was proposed to use tokens of different colors. It is some additional data that is stored on the token's stack. The fork nodes generate a unique color every time a token passes through it. The output tokens have the same color, it means that the fork's color is placed on the token's stack of colors. For join node activation it is necessary that the colors at the top of the stacks have the same color. If the condition is right the join node becomes activated, and the output token remains all colors except the top color. In another case, a mistake occurs and verification is completed.

The list of tracked model's mistakes are presented.

1) Lexical: The signature starts with a small letter; the activity's name begins with a verb; the decision does not have a question mark; the alternative is not signed; the flow has a signature, not being an alternative or a condition; the use of an element that does not belong to AD; the use of special characters in the naming.

2) Syntactic: There is no initial node or no final node, or no action nodes; more than one initial node; several decision nodes in a row are used; the number of incoming or outgoing flows does not match the required one; the element does not belong to any participant; the name of the action node, or participant is not unique; alternatives lead to the same element; unpaired use of a fork and join nodes; the use of an empty swimlane.

3) Semantic: Decision's alternatives are the same.

Fig. 3 and Fig. 4 present possible AD students mistakes.

Fig. 3. The example of AD mistake – inapproper number of output flows.

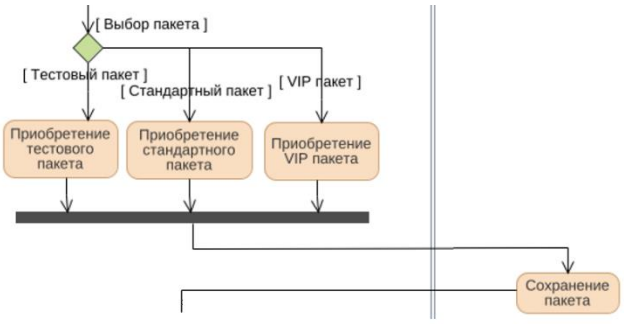

Fig. 4. The example of AD mistake – impossibility to activate join node.

E. Approach for Class Diagram Verification

There are very few fully automated methods for verifying CD. Most of the existing solutions require a translation process into specific data formats that must be performed by a human. This approach is not suitable, since we need to quickly verify the diagrams, and the translation process takes a sufficient amount of time.

The verification process is based on and similar to the program compilation analysis stage, and it can be divided into three stages: the first stage is lexical analysis, the second is syntactic analysis and the third stage is semantic analysis. At each stage, the corresponding rules will be checked. During the first stage, metadata is converted into a set of tokens, the use of invalid tokens, incorrect names, designations and properties of tokens is detected. During the second stage, the correctness of creating constructions of the UML language from a valid set of tokens. And at the final stage of verification, the semantics of the constructed class diagram is considered, namely the correctness of the semantic meanings of words, phrases and elements.

Verification process begins with reading all data about the model from the XMI file. All properties of the CD tokens can be retrieved from these data. Already on the basis of these properties, it will be easy to detect some inconsistencies and mistakes.

The main point in this method is to designate a set of rules for constructing UML CD such as to identify all mistakes in the verified diagram. The set of rules was compiled using the UML specification [1]. The list of all rules that will be checked during verification was described in detail earlier in Chapter 1. Also, special attention will be paid to common mistakes when constructing CD.

At the moment, the verifier is able to find the following lexical, syntactic and semantic errors in CD.

1) Lexical: Using tokens that are not allowed for the class diagram (actor, use case, component). Incorrect class, interface or enumeration name (it must begin with a capital letter and must not contain spaces). Incorrect attribute or operation name (it must begin with a lowercase letter (except constructors and destructors) and must not contain spaces). Data type names do not match the names used in the target programming language or other class names of the model. The text of various restrictions is not enclosed in curly braces {}.

2) Syntactic: There are classes, interfaces and enumerations that do not have connections. During

composition the multiplicity from the composite object exceeds one. During composition or aggregation at least one of the main object attributes type contains the name of the subordinate object. The text of restrictions for a class do not match its attributes. The class containing protected members or operations is not associated with its descendants. Between the element "class" and "enumeration" there is not a "dependency" relationship.

3) Semantic: When specifying the roles multiplicity, some numbers are negative integers. If multiplicity is indicated as interval, it does not begin with a smaller number.

F. Verification System for Three Types of UML Diagrams

The system has the following features:

1) the ability to upload one or several files into the system;

2) the ability to automatically find the pair "metadata - image" while files are uploading into the system;

3) the ability to add and remove diagrams from the current list of models;

4) the ability to work only with metadata diagrams (without images);

5) dynamic changing the graphical presentation of diagrams while switching is occurring between them;

6) dynamic mistakes designation on the graphical presentation of the diagram while switching is occurring between errors;

7) highlighting each error in a different color depending on its severity;

8) the ability to verify all diagrams with "not verified" status at once.

IV. Realization

The process of verification system creating was divided into two stages.

1) Implementation of the UCD, AD and CD verification modules in prototype mode as a console application and presentation the result in the form of text message: C# (UCD and CD verification modules) and Java (AD verification module) were used for implementation. At this stage, we tested the detection of the most common mistakes of groups 1 and 2.

2) Integration of UCD, AD and CD verification modules into a system with a user graphical interface and realization of package processing function: The AD verification module at this stage was implemented in C#. The detection of new mistakes of groups 1 and 2, which appeared in the works of students, and mistakes of group 3 were tested.

Fig. 5 shows the results of CD verification.

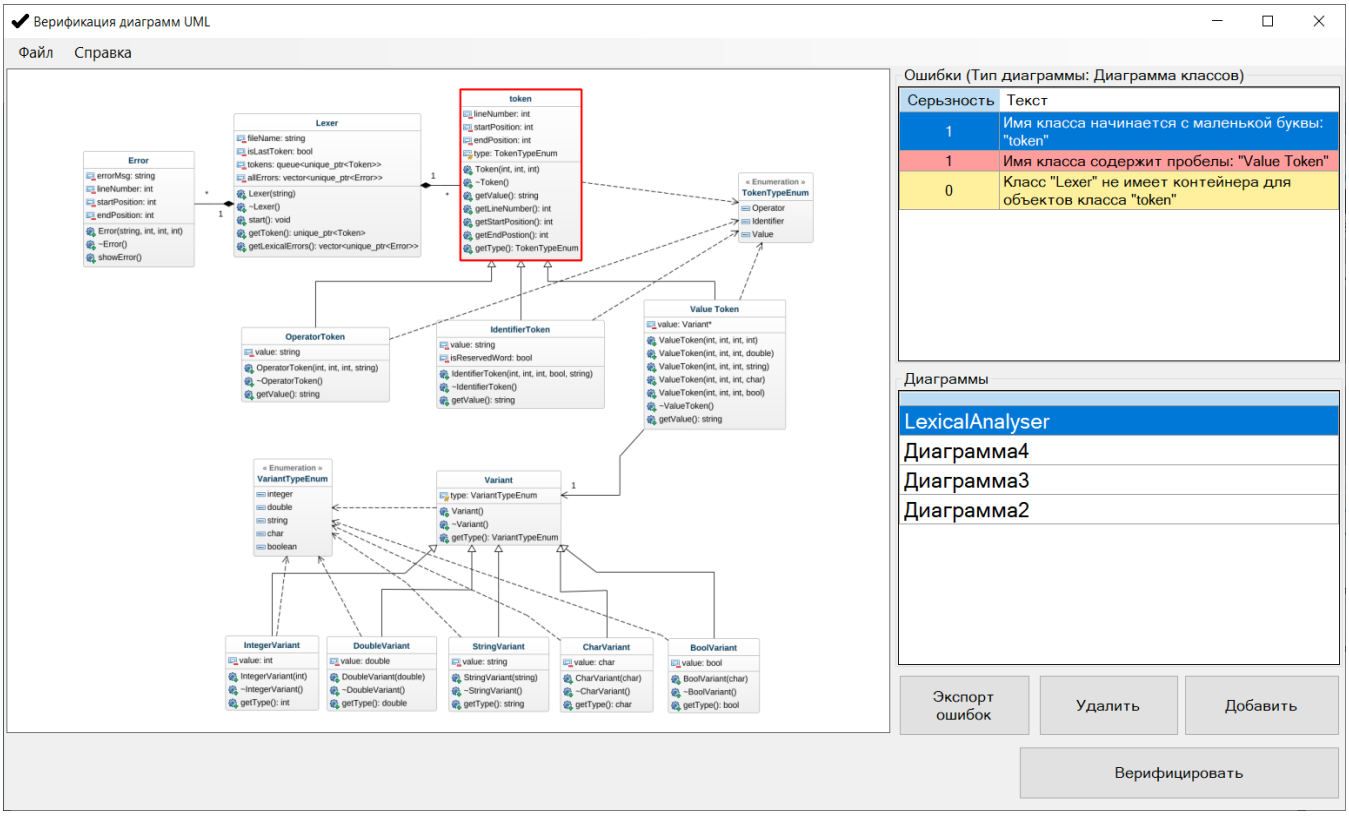

Fig. 5. Results of CD verification.

V. Results

The UCD verification module at the prototype stage was tested on 70 student models. The list of initially identified 25 errors in the process of integration into the system was expanded with the following errors: the lack of a hierarchy of use case relationships, the lack of a package element, the use of elements of other diagram types. In the AD verification module, at the prototype stage, 19 types of errors were detected using the example of 30 student models. The initial list of mistakes during integration into the system was supplemented with the use of special characters in naming, the absence of an initial or final state, and the use of an empty swimlane. 80 errors CD verification module detects were tested on an example of 26 students' works. All of them are presented in the list of errors that are processed by the verifier in the integrated system.

VI. Discussion and Future Work

The current version of the UML diagram verification system solves the following tasks:

1) First: Based on the exported XMI file, mistakes are searched in AD, ACD and CD.

2) Second: Visualization of found mistakes and their display on exported diagram images.

It can be recommended for use by two categories of users:

1) Students: To check models before submitting for teacher's grading.

2) Teachers: For verification of diagrams in package mode.

For the future we will work on the validation functions for teachers in order to qualify the degree of deviation of the student model from the task specification and to form the recommended score for the model.

VII. Conclusion

139 student's UCD models, 41 AD models, and 26 CD models were analyzed. A classification of diagram's mistakes into two groups, model mistakes and positioning mistakes, was proposed. The choice of the tool for creating UML diagrams was justified. The lexical, syntactic and semantic analysis of the metadata of the UCD, AD and CD models exported from GenMyModel was performed. Modules for verification of three types of UML diagrams were developed and realized. These modules were integrated into a system allowing package processing of model files.

References

[1] Unified Modeling Language 2.5.1. (2017). Object Management Group. Accessed: Mar. 31, 2021. [Online]. Available: https://www.omg.org/spec/UML/About-UML/

[2] D. Boberic and D. Tesendic, "Experience in Teaching OOAD to Various Students," Informat. Educ., vol. 12, pp. 43-58, 2013.

[3] L. Baresi, A. Morzenti, A. Motta, M. M. Pourhashem K., and M. Rossi, "A Logic-Based Approach for the Verification of UML Timed Models," ACM Trans.Softw. Eng. Methodol., vol. 26, no. 2, article 7, Oct. 2017.

[4] Z. Daw, John Mangino, and R. Cleaveland, "UML-VT: A Formal Verification Environment for UML Activity Diagrams," in Proc. P&D@MoDELS, 2015. [Online]. Available: http://ceur-ws.org/Vol-1554/PD_MoDELS_2015_paper_16.pdf.

[5] P. Bourque, R. Dupuis, A. Abran, J. W. Moore, and L. Tripp, "Guide to the software engineering body of knowledge," IEEE Computer Society Press, 1999.

[6] MDA Guide rev. 2.0 (2014). [Online]. Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01

[7] R. Reuter, T. Stark, Y. Sedelmaier, D. Landes, J. Mottok, and C. Wolff, "Insights in Students' Problems during UML Modeling," in 2020 IEEE Global Eng. Educ. Conf. (EDUCON), Porto, Portugal, pp. 592-600.

[8] K. Matyokurehwa and K. T. Makoni. "Students' Perceptions in Software Modelling Using UML in Undergraduate Software Engineering Projects," Int. J. Inf. Commun. Technol. Educ., vol. 15, no. 4, 2019.

[9] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and M. Pourzandi, "Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages," Electron. Notes Theor. Comput. Sci., vol. 254, pp. 143–160, 2009.

[10] S. Chren, B. Buhnova, M. Macak, L. Daubner, and B. Rossi, "Mistakes in UML Diagrams: Analysis of Student Projects in a Software Engineering Course," in Proc. 2019 IEEE/ACM 41st Int. Conf. Softw. Eng.: Softw. Eng. Educ. and Training (ICSE-SEET), Montreal, QC, Canada, pp. 100-109.

[11] A. M. Fernández-Sáez, D. Caivano, M. Genero, and M. R. V. Chaudron, "On the use of UML documentation in software maintenance: Results from a survey in industry," in Proc. 2015 ACM/IEEE 18th Int. Conf. Model Driven Eng. Languages and Syst. (MODELS), Ottawa, ON, Canada, pp. 292-301.

[12] 5 Common UML Mistakes. (2014). GenMyModel [Online]. Available: http://blog.genmymodel.com/5-common-uml-mistakes.html

[13] K. Chytalová, 2018, "Katalog chyb v UML diagramech PB007 - Softwarové inženýrství I," (in Czech) [Catalog of errors in UML diagrams PB007 - software engineering I], Lasaris Lab, Faculty of Informatics, Masaryk University [Online]. Available: https://drive.google.com/file/d/1J3_Ueb4E2YdAZjksryC4-F123Xqmyhkm/view

[14] J. Campbell. (2007). Verification and Validation. UML Models [PowerPoint slides]. Available: https://docplayer.net/24240668-Uml-models-lecture-10-part-1-verification-and-validation-uml-models-2-non-uml-models-verification-and-validation.html

[15] F. Mokhati, P. Gagnon, and M. Badri, "Verifying UML Diagrams with Model Checking: A Rewriting Logic Based Approach," Seventh Int. Conf. Quality Softw. (QSIC), Portland, OR, USA, 2007, pp. 356-362.

[16] J. Cabot, R. Claris 'o, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," 2008 IEEE Int. Conf. Softw. Testing Verification and Validation Workshop, Lillehammer, Norway, 2008, pp. 73-80.

[17] A. Delgado, A. Dias, and F. Brito e Abreu, Verification and Validation of UML Diagrams using Checklists [Online]. Available: http://docplayer.net/33548062-Verification-and-validation-of-uml-diagrams-using-checklists.html [18] Dresden OCL. (2016). Accessed: Mar. 31, 2021. [Online]. Available: https://github.com/dresden-ocl/dresdenocl

[18] XML Metadata Interchange (XMI) Specification 2.5.1. (2015). Object Management Group. Accessed: Mar. 31, 2021. [Online]. https://www.omg.org/spec/XMI/2.5.1/PDF

[19] R. Conradi, P. Mohagheghi, T. Arif, L. C. Hegde, G. A. Bunde, and A. Pedersen, "Object-Oriented Reading Techniques for Inspection of UML Models – An Industrial Experiment," in Lecture Notes in Computer Science, vol. 2743, Berlin, Heidelberg: Springer, 2003.

[20] G. Kösters, H. Six, and M. Winter, Validation and Verification of Use Cases and Class Models [Online]. Available: https://www.researchgate.net/publication/2330184_Validation_and_Verification_of_Use_Cases_and_Class_Models

[21] L. Baresi and M. Pezzè, "On Formalizing UML with High-Level Petri Nets," in Concurrent Object-Oriented Programming and Petri Nets. Lecture Notes in Computer Science, vol. 2001, Berlin, Heidelberg, Springer, 2001.

[22] J. Araujo and A. Moreira, "Integrating UML Activity Diagrams with Temporal Logic Expressions," in Proc. 10th Int. Workshop on Exploring Modeling Methods in Syst. Anal. and Design (EMMSAD), 2005. [Online]. Available: http://ceur-ws.org/Vol-363/paper8.pdf [24] A. Raschke, "Translation of UML 2 Activity Diagrams into Finite State Machines for Model Checking," in Proc. 2009 35th Euromicro Conf. on Softw. Eng. and Advanced Appl., Patras, Greece, pp. 149-154.

[23] V. Rafe and A. T. Rahmani, "Formal Analysis of Workflows Using UML 2.0 Activities and Graph Transformation Systems," in Lecture Notes in Computer Science, vol. 5160, J.S. Fitzgerald, A.E. Haxthausen, and H. Yenigun, Eds., 2008, pp. 305–318.