Automated Change Propagation from Source Code to Sequence Diagrams

Karol Rástočný(✉) and Andrej Mlynčár

Faculty of Informatics and Information Technologies,
Slovak University of Technology in Bratislava,
Ilkovičova 2, 842 16 Bratislava, Slovakia
karol.rastocny@stuba.sk, a.mlyncar@gmail.com

Abstract. Sequence diagrams belong to three most frequently used UML diagrams and they are often an integral part of a software design. Designers utilize sequence diagrams to define and visualize designed software's behavior. But during software development and maintenance, multiple vendor's changes are implemented into a source code. These changes lead to inconsistencies between a software model and the source code, that are omitted due to lack of time. This paper is focused on problems with automated source code changes propagation into UML sequence diagrams. In the paper, we propose the architecture for synchronization of outdated designers' sequence diagrams with current software behavior implemented in a source code. The proposed architecture is focused on updating and not on regenerating sequence diagrams, what helps designers to understand modified behavior and changes provided in it. We evaluated the proposed architecture via implemented extension for Eclipse Papyrus, which analyzes differences between sequence diagrams and source code model, and based on developers' styles, it propagates differences to sequence diagrams.

Keywords: UML · Sequence diagram · Source code · Change propagation

1 Introduction

With the focus on current trends in agile software development, great emphasis is placed on software sustainability. Huge amount of change requests is often required in massive enterprise architectures and long-term projects. Even though change requests and bugs are obviously well documented [1], during applying of these changes, there is regularly a problem that software design documentation is not properly updated and become insufficient [1, 2]. Outdated software design documentation, e.g. in form of a UML model can cause significant obstacles during problem investigation.

To solve this problem, automated tools for change propagation from source code to UML models are required. Tools for automated synchronization of static part of UML models are already developed and integrated in software modelling tools, e.g. Sparx Systems Enterprise Architect[1] or IBM Rational Software Architect Designer[2]. But

[1] http://www.sparxsystems.com/.
[2] http://www-03.ibm.com/software/products/en/ratsadesigner.

A M. Tjoa et al. (Eds.): SOFSEM 2018, LNCS 10706, pp. 168–179, 2018.
https://doi.org/10.1007/978-3-319-73117-9_12

software behavior is still uncovered problem. There are already tools (e.g. IBM Rational Rhapsody[3] or Microsoft Visual Studio[4]) and research works [3, 4] that are dealing with generating behavioral UML diagrams from source code. But these tools can only generate new diagrams from source code or a source code execution, but they are not able to update existing diagrams with changes provided in source code. This problem is mainly visible in the third most used UML diagram [5, 6] – sequence diagram, in which designers obviously keep needed abstraction and they do not model all scenarios and interactions. The stable position of sequence diagrams in software models over years is caused by their ability to clarify how software works [5].

To achieve automatic change propagation to UML sequence diagrams, a source code and sequence diagrams synchronization is required. This means that if we want to preserve meaning and level of abstraction of models, every time changes are made in a source code, existing sequence diagrams should be updated. There is also problem with level of abstraction of sequence diagrams, because transforming all source code changes is usually not desired. We propose the solution fully functional synchronization of software source code and sequence diagrams based on change detection and synchronization methods.

2 Related Work

To achieve fully functional synchronization process of UML sequence diagrams and source code, it is required that sequence diagrams' components and source code fragments, that can be transformed into sequence diagrams, need to be represented in identical form suitable for detection of changes. One of promising sequence diagram representation is a hierarchical tree structure [7], where tree node represents sequence diagram lifeline and sequence diagram messages are represented with edge of tree (Fig. 1).

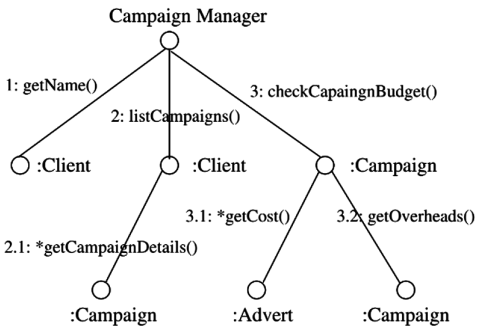

Fig. 1. Example of hierarchical tree representation of a sequence diagram [7]

[3] http://www-03.ibm.com/software/products/en/ratirhapfami.

[4] https://www.visualstudio.com/.

Another example of solutions where diagram was transformed into suitable structure for later processing and comparison are Petri nets [9] or control flow graphs [8].

An important part of the synchronization is source code fragments extraction. These fragments are required for comparison and synchronization of source code and a set of sequence diagrams. Source code fragments extraction can be made during source code execution [13] or by manual analysis of program files. Another way to extract information about program structure is static analysis with source code transformation to Abstract Syntax Tree [3] or Knowledge Discovery Metamodel (KDM)[5]. Knowledge Discovery Metamodel is technology independent metamodel developed by OMG, usually used in legacy system to provide intermediate representation of software components and software structure. KDM is separated into 4 layers – Infrastructure layer, Program Elements Layer, Runtime Resource Layer and Abstractions Layer [10]. Program Elements Layer provide us with information about source code structure in XMI format. MoDisco Eclipse Plugin implements KDM standard and it generates KDM XMI structure from Java Standard Edition projects [12].

3 Architecture for Automated Change Propagation

Our proposed solution of automated change propagation from source code to sequence diagrams is designed as modular architecture containing seven modules that expose services (see Fig. 2):

- *KDM Code Analyzer* – analyzes source code described by KDM and transforms it to an object model;
- *UML Analyzer* – analyzes UML model, extracts information about sequence diagrams and transforms them to an object model;
- *Strategy Analyzer* – analyzes design strategies used by designers in sequence diagrams and prepares data for synchronization rules;
- *Graph Transformation module* – transforms source code object model and sequence diagram object model to comparable graph representations;
- *Comparison module* – compares the source code graph and sequence diagrams graphs and builds a list of changes provided in the source code;
- *Synchronization module* – uses the list of detected changes and synchronizations rules to propagate changes into a changelog, which should be applied on sequence diagrams;
- *Interpreter module* – interprets the changelog on the UML model.

The solution can extract information about sequence diagrams from a software UML model and source code fragments generated by MoDisco Eclipse Plugin from source code and use this information to synchronize newly added modifications in source code into existing sequence diagrams.

Each module has defined output and input formats and acts mostly independently of the other modules. This means that even if a core functionality in one of modules has

[5] http://www.omg.org/technology/kdm/.

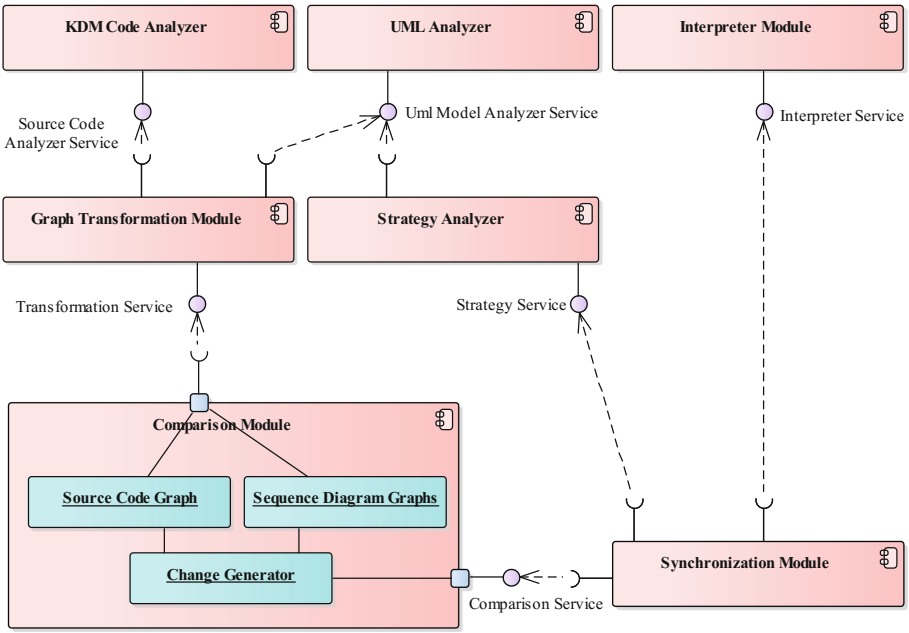

Fig. 2. Modular architecture for automated change propagation from source code to sequence diagrams.

been changed, inputs and outputs of the module remain unchanged so functionality of the rest of modules is sustained. For example, if the *KDM Analyzer* module is replaced with an AST analyzer, it should not affect functionality of other modules. Another example is in comparison module – if another comparison algorithm is used in comparison module, this change does not affect ways of a source code and sequence diagrams synchronization.

3.1 UML Analyzer

The module *UML Analyzer* parses an UML model stored in a XMI file and transforms the model's sequence diagrams to simplified sequence diagram representation (Fig. 3), which contains necessary data for comparing newer source code with outdated sequence diagrams. This simplified representation is efficient for later transformations and it also suitable for representation of algorithms written in the source code.

3.2 KDM Analyzer

The *KDM Analyzer* provides an adapter between the proposed architecture and used tool for static source code analysis. Getting the *KDM Analyzer* apart makes the architecture programming language and technology independent. We can easily implement language specific adapter, and only by implementation of specialized adapter, we can make IDE specific implementation, while the specialized adapter can

reuse source code model of an IDE (e.g., CodeModel in case of Visual Studio). These IDE specific implementations should be more efficient and accurate.

The *KDM Analyzer* transforms source code's model in KDM to the common representation as the *UML Analyzer*. Re-usage of the representations adds more logic to the *KDM Analyzer*, what is not ideal for adapters. But nearness to a source code gives better possibilities for efficient mapping source code artifacts to sequence diagram artifacts and it reduce complexity of later source code and sequence diagrams comparison.

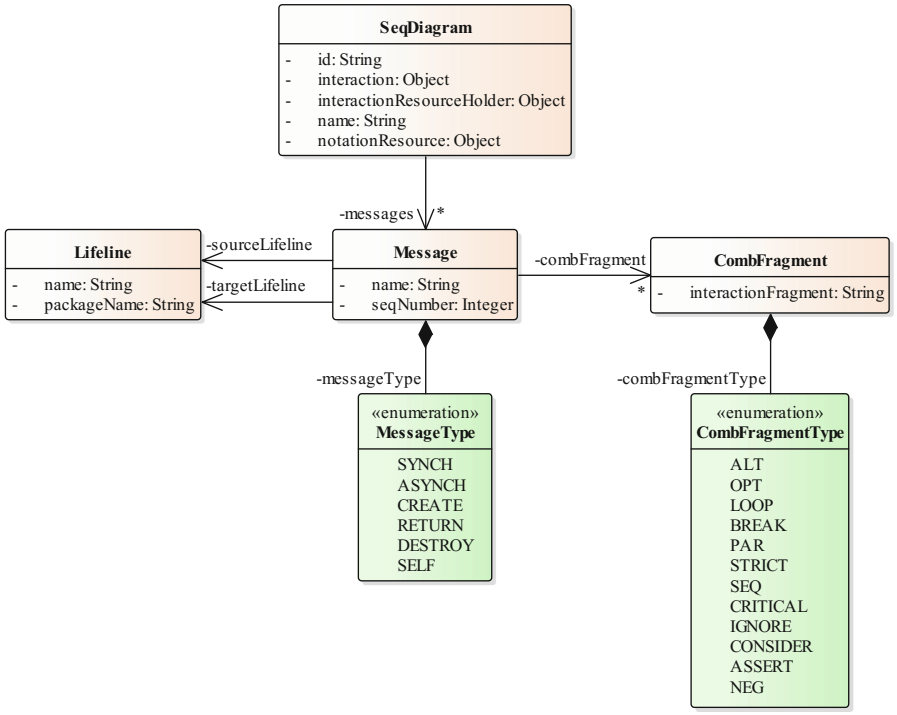

Fig. 3. Simplified sequence diagram representation generated by the *UML analyzer.*

3.3 Graph Transformation Module

The main objective of the *Graph Transformation module* is to transform data created by the *KDM Analyzer* and the *UML Analyzer* to a format which supports efficient change detection between a source code and sequence diagrams. There are several representation solutions mentioned in the Sect. 2. Based on analysis and comparison of these methods we use *Hierarchical tree structure* [7], based on which we proposed our *sequence diagram tree graph structure*. Each tree node (class *Node*) contains lifeline name, execution identifier, list of child nodes, message by which node is created, and list of combined fragments affecting this message.

3.4 Comparison Module

The *Comparison module* detects changes between a source code tree structure and sequence diagram tree structures created by the *Graph Transformation module*. The *Comparison module* process each potentially outdated sequence diagram in three steps:

1. *Find identical subtree* in the source code tree structure – the module tries to match the sequence diagram in the source code. If the sequence diagram is found as identical subtree, the diagram is marked as up-to-date and the next two steps are omitted.
2. *Find the root node of similar subtree* in the source code tree structure – the module tries to identify the root node of the sequence diagram in the source code. To match the root node, the breadth-first search with following conditions is used:
   a. For a root node candidate, a subtree-depth is calculated. If the subtree depth is less than the sequence diagram's tree-depth - 2, the root node candidate is rejected. The difference 2 has been chosen as search algorithm optimization, while it markedly reduces searched space. We defined this heuristic, because the source code is more detailed than sequence diagrams and if the source code's subtree is significantly smaller, than it will been rejected in the following condition with high probability.
   b. If 35% nodes of the root node candidate's subtree are identical with the sequence diagram tree's nodes, the root node candidate is marked as the root node. The ratio of identical nodes has been determined by manual experiments. The value 35% is relatively small, but the source code tree contains precise details like system and external calls at lowest level, that are not modeled in sequence diagrams.
3. *Comparison of the source code's subtree with the sequence diagram tree* – the comparison algorithm is based on [11], which finds an edit script which contains a set of tree modifications to achieve quickest way to reach an isomorphic state between these two trees. In our case, our modified change detection algorithm creates a list of changes detected in compared source code's subtree and the sequence diagram tree. Detected changes are later processed by synchronization module to finalize synchronization process.

The comparison algorithm is proposed to process a source code representable by a tree, i.e. programs with one execution point (e.g. *Main* function). The algorithm can be modified to reflect a source code with multiple execution points, e.g. REST services. In this case, the source code is not a tree, but it is still directed graph with nodes that have not any input edges. These nodes are execution points and they acts as root nodes for their sub-graphs – trees. Therefore, by matching sequence diagrams against all root nodes (execution points), the comparison algorithm became suitable to current software systems that utilize execution frameworks.

3.5 Strategy Analyzer

The *Strategy Analyzer module* analyzes original sequence diagrams and based on this analysis it collects information, that describe a sequence diagrams design style. Current set of analyzed design style information contains:

- Average lifelines count;
- Average messages count;
- Lifelines count in each sequence diagram;
- Messages count in each sequence diagram;
- Frequencies of combined fragment types;
- Usage of get/set messages in each sequence diagram.

3.6 Synchronization Module

The *Synchronization module* is responsible for managing synchronization process. From this module, users can start execution of synchronization process. The module uses lists of changes and sets of design style information for each sequence diagram to build a list of synchronization actions, that should be provided in the UML model.

The list of synchronization actions is built by resolutions, whether a change detected in the *Comparison module* should be interpreted to a software sequence diagram. This feature is fulfilled by set of synchronization rules. Each modification detected in the *Comparison module* is evaluated by following synchronization rules and based on output from rule, it is determined, if the modification is added to the list of synchronization actions:

- Lifeline synchronization rules:
  - if an addition of a lifeline exceeded maximal lifelines count in the sequence diagram, the addition is ignored;
  - if an addition of a lifeline will introduce a package, which has not been used in the sequence diagram, the addition is ignored;
- Messages lifeline synchronization rules:
  - If multiple occurrences of a message should be added to the sequence diagram, but the sequence diagram does not contain any occurrence of the message, all additions of the message are ignored;
  - If the sequence diagram does not contain any get/set messages, all additions of get/set messages are ignored;
- Combined fragments synchronization rules:
  - If multiple combined fragments should be added, but the sequence diagram does not contain any combined fragment, the sequence diagram is evaluated as high level diagram and all additions of combined fragments are ignored;
  - If an opt combined fragment should be deleted and alt combined fragment should be added at the same position, both modifications are ignored and new modify synchronization action, which transforms the opt combined fragment to the alt combined fragment, is added to the list of synchronization actions.

3.7 Interpreter Module

The *Interpreter module* is designed to finalize whole synchronization process. The module interprets actions from the list of synchronizations actions. The *Interpreter module* should be implemented for each UML modelling tool separately. There is also possibility to provide tool independent implementation that modifies XMI files, but each UML modelling tool uses their own XMI extension for sequence diagrams' layout information.

4 Evaluation

To evaluate usability of the proposed architecture we implemented the prototype[6] which synchronizes Java source code analyzed by MoDisco Eclipse Plugin with sequence diagrams modeled in Eclipse Papyrus. In the prototype, we did not implement synchronization of all elements from the sequence diagram metamodel, but we focused on the mainly used elements, via which we can present correctness of change propagation from obviously used source code structures:

- Synchronous messages;
- Reply messages;
- Lifelines;
- Combined fragments: loop, opt.

Modules of the prototype are implemented as OSGi Eclipse Bundles with fully implemented APIs and data structures necessary for supporting whole sequence diagram metamodel. So, this restriction of sequence diagram elements does not affect results of the evaluation and the restriction will be resolved by final implementation of the change detection module and the interpreter module.

We evaluated the proposed architecture via sixteen test cases, that was organized in three test sets based on level of their complexity:

- Evaluation of basic functionalities
  - TC01: Adding a synchronous message
  - TC02: Adding a synchronous message and a lifeline
  - TC03: Removing a synchronous message
  - TC04: Removing a synchronous message and a lifeline
  - TC05: Adding a combined fragment opt
  - TC06: Removing a combined fragment opt
- Evaluation of synchronization rules
  - TC07: Filtration of system calls
  - TC08: Restriction of lifelines count
  - TC09: Filtration of get/set calls
  - TC10: Filtration of external calls
  - TC11: Filtration of combined fragments

[6] Replication package: https://github.com/rastocny/SOFSEM_SeqDiag_ChangeProp.

- Evaluation of propagation of complex changes
  - TC12: Replacing two messages with one new message, which contains internally six new calls
  - TC13: Condition change and movement of existing calls to new operation
  - TC14: Part of the functionality has been moved to new operation
  - TC15: Removing a sequence diagram implementation from the source code
  - TC16: Adding a loop over an existing condition and adding a new synchronous call into the condition

For each test case, we defined outdated sequence diagram (Fig. 4), modified source code (List. 1) and expected changelog (List. 2). After execution of all test cases we manually compared expected changelogs with obtained changelogs and evaluated differences. In the next step, we reviewed updated sequence diagrams (Fig. 5) and evaluated their layout and correctness.

During the evaluation of the first test case set, 20 modifications in sequence diagrams were done. The evaluation proved, that the prototype correctly processes the source code and sequence diagrams in *UML Analyzer* and *KDM Analyzer* modules and that the source code and the sequence diagrams are correctly transformed to the *sequence diagram tree graph structure*. This evaluation also showed, that the *Comparison module* can detect modifications on implemented sequence diagram elements and that detected modifications are correctly interpreted by the *Interpreter module*.

The first set of test cases uncover some layout issues in the *Interpreter module*. After deletion of messages, bellow messages are not shifted up. There was also problem with added combined fragment which has not correctly set top and bottom margins.

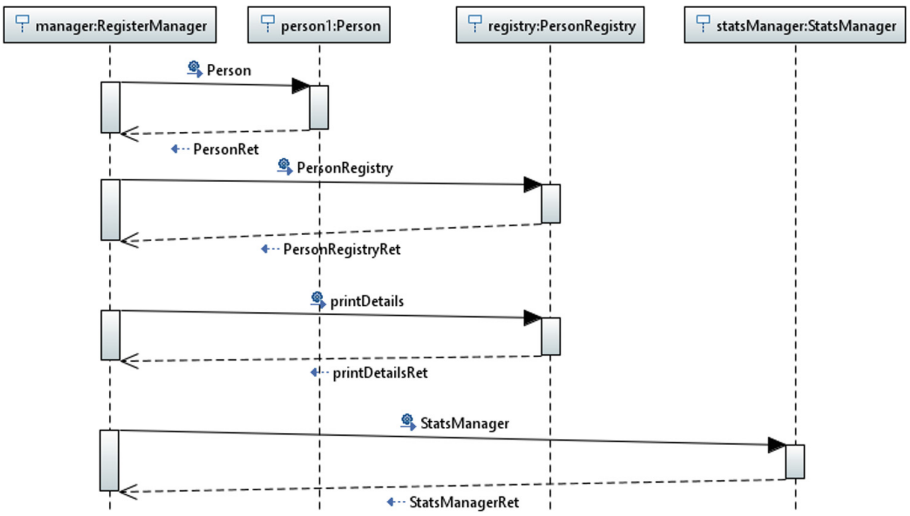

Fig. 4. Outdated sequence diagram for testcases TC01-TC07.

```
public void createRegistry(){
  Person person1 = new Person("Andrej", "Mlyncar", null);
  PersonRegistry registry = new PersonRegistry();
  if(registry != null)
    registry.printDetails();
  StatsManager statsManager = new StatsManager(registry);
}
```

List. 1. Modified source code for TC03.

```
fragment_add = opt:registry!=null; message: printDetails
```

List. 2. Expected change log for TC03.

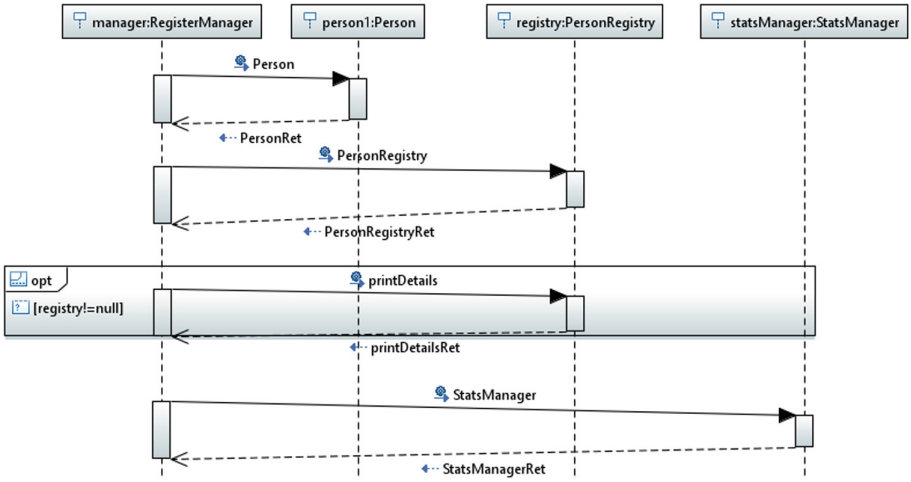

Fig. 5. Updated sequence diagram for TC03.

The second test case set is focused no validation if the proposed architecture is able to detect basics of used design styles. For these test cases, we defined different sequence diagrams and we observed if the *Strategy Analyzer* and the *Synchronization module* correctly detect and interpret used styles. The test cases applied 20 modifications in sequence diagrams and proved that observed design styles are correctly identified independently. We observed only one new issue with calculating horizontal positions of nested execution specifications.

Combinations of multiple design styles and source code modifications were evaluated by the last test case set, which provided 67 modifications. The results showed that proposed architecture can correctly detects complex changes and propagates them to sequence diagrams with respect of a design style. Some issues were observed in layouts

of sequence diagrams, where multiple modifications have been done. Some sequence diagram elements did not have correctly calculated heights and x-coordinates, but all elements were semantically and syntactically placed correctly. These problems can be later resolved by reusing Eclipse Papyrus's layouting algorithm in the *Interpreter module*. The layouting algorithm should be also updated to use distances and sizes learned from original sequence diagrams for added and modified elements.

5 Conclusion and Future Work

The work presented in the paper is primarily focused on architectural design of solution which will be able to provide an effective way to automate update of a behavioral documentation of software systems for software architects and developers. The proposed solution can improve process of applying changes to existing software systems by reducing communication about implemented software changes between software developers and architects or analytics.

Synchronization of source code and sequence diagrams is executed by set of modules. Modules operate independently of other modules functionality, which means that interpretation, comparison or synchronization methods can be changed without any or significant effects to other modules and that core functionality is language and tool independent.

We implemented the first prototype to prove concepts of the proposed architecture and to evaluate its applicability. The next steps are focused on completion of the implementation with support of all applicable sequence diagram elements. After that we will provide final evaluation of the architecture in two steps. Firstly, we will utilize modularity and we will implement a sandbox system which replaces the *Graph Transformation module* and the *Interpreter module*. The sandbox system will test robustness of the solution with generating test cases and observing results of the *Synchronization module.* In the second step, we will provide empirical study by applying the implemented prototype in real agile teams. In this study, we will deploy the prototype into the tool for collaborative modelling [14] and we will involve teams from the course Team project and teams from our innovation lab built in cooperation with the project DA-SPACE[7].

Later we will focus also on applying the proposed architecture on other behavioral diagrams. We assume that the architecture is almost directly applicable on communication diagrams, that have equivalent expression power as sequence diagrams. More challenging are activity diagrams, in that same algorithmic concepts (e.g., loops) can be modelled variously.

Acknowledgement. This work was partially supported by the Scientific Grant Agency of the Slovak Republic, grant No. VG 1/0752/14, the Slovak Research and Development Agency under the contract No. APVV-15-0508, and this publication is the partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

[7] http://www.interreg-danube.eu/approved-projects/da-space.

References

1. Voigt, S., von Garrel, J., Müller, J., Wirth, D.: A study of documentation in agile software projects. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, p. 6. ACM, New York (2016)
2. Rashid, N., Khan, S.: Developing green and sustainable software using agile methods in global software development: risk factors for vendors. In: Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering, pp. 247–253. SCITEPRESS (2016)
3. Fauzi, E., Hendradjaya, B., Sunindyo, W.D.: Reverse engineering of source code to sequence diagram using abstract syntax tree. In: International Conference on Data and Software Engineering (ICoDSE), p. 6. IEEE (2016)
4. Srinivasan, M., Yang, J., Lee, Y.: Case studies of optimized sequence diagram for program comprehension. In: 24th International Conference on Program Comprehension (ICPC), p. 4. IEEE (2016)
5. Dobing, B., Parsons, J.: How UML is used. Commun. ACM - Two Decades Lang-action Perspect. **49**(5), 109–113 (2006)
6. Reggio, G., Leotta, M., Ricca, F., Clerissi, D.: What are the used UML diagram constructs? A document and tool analysis study covering activity and use case diagrams. In: Hammoudi, S., Pires, L.F., Filipe, J., das Neves, R.C. (eds.) MODELSWARD 2014. CCIS, vol. 506, pp. 66–83. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25156-1_5
7. Li, X., Liu, Z., Jifeng, H.: A formal semantics of UML sequence diagram. In: Australian Software Engineering Conference 2004, pp. 1–10. IEEE (2004)
8. Rountev, A., Volgin, O., Reddoch. M.: Control flow analysis for reverse engineering of sequence diagrams. Technical report, Ohio State University (2004)
9. Emadi, S., Shams, F.: Transformation of usecase and sequence diagrams to petri nets. In: ISECS International Colloquium on Computing, Communication, Control, and Management 2009, pp. 399–403. IEEE (2009)
10. Pérez-Castillo, R., De Guzman, I.G.R., Piattini, M.: Knowledge discovery metamodel-ISO/IEC 19506: a standard to modernize legacy systems. Comput. Stan. Interfaces **33**(6), 519–532 (2011)
11. Wang, Y., DeWitt, D.J., Cai., J.-Y.: X-Diff: an effective change detection algorithm for XML documents. In: 19th International Conference on Data Engineering, pp. 519–530. IEEE (2003)
12. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 173–174. ACM, New York (2010)
13. Oechsle, R., Schmitt, T.: JAVAVIS: automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). In: Diehl, S. (ed.) Software Visualization. LNCS, vol. 2269, pp. 176–190. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45875-1_14
14. Ferenc, M., Polasek, I., Vincúr, J.: Collaborative modeling and visualisation of software systems using multidimensional UML, In: Proceedings of the Fifth IEEE Working Conference on Software Visualization VISSOFT 2017, p. 5. IEEE, Shanghai (2017)