Enhancing UML Model Dynamics: A Source Code Generation Approach

Michal Staňo[1], Matej Čiernik[2], Ladislav Hluchý[1], Martin Bobák[1] and Jean Rosemond Dora[1]

[1] Institute of Informatics, Slovak Academy of Sciences, Bratislava, Slovakia
{michal.stano, ladislav.hluchy, martin.bobak, jeanrosemond.dora}@savba.sk
[2] Touch4IT, Bratislava, Slovakia; matej.ciernik@touch4it.com

Abstract: This work aimed to develop functionality for generating source code from dynamic models, specifically, UML sequence diagrams, using the Object Action Language (OAL). The existing prototype, AnimArch, will use the generated source code to provide animation of UML model dynamics. The paper includes the theoretical background of software modelling, the technologies applied, and reviews of relevant existing publications. This paper also contains a study focused on evaluating the final solution, its practical applicability, and the relevance of our approach in generating OAL source code from sequence diagrams. The following results demonstrate the efficiency and practical utility of the developed solution in enhancing the capabilities of the AnimArch prototype.

Keywords: code synthesis; UML sequence diagram; Object Action Language; model-driven development; automated code generation; model-based design; software modelling; dynamic model transformation

1 Introduction

The primary objective of this research is to develop functionality that enables the generation of source code from dynamic models, specifically UML sequence diagrams, using the Object Action Language (OAL). This generated code will be integrated into the AnimArch prototype, a tool designed to support the animation of UML model dynamics.

Model-driven development (MDD) promotes the creation and use of domain models, which developers generate to automate the production of software artefacts, thereby increasing software productivity and maintainability. In this context, xtUML and OAL are crucial as they provide platform-independent action definitions and facilitate the translation of abstract models into executable code [1].

In this study, we present the technologies used and evaluate the practicality of our solution. The research project aims to highlight the results of generating source code from dynamic models and to inspire consideration of ways to assist users of the *AnimArch* prototype.

2 Related Work

Significant effort has been devoted to transforming dynamic models into source code in model-driven development (MDD). This section outlines our approach's main related works and methodologies, explicitly converting sequence diagrams into Object Action Language (OAL) code within the Executable and Translatable UML (xtUML) framework. We also review related publications that partially overlap with our research objectives.

Model-driven development (MDD) is an approach to software development focused on creating and utilizing domain models to automate code generation and other software artefacts. The primary goal of MDD is to increase software productivity and maintainability through the abstraction and automation of repetitive and non-technical programming activities [2-3].

Executable and Translatable UML (xtUML) is a language for model-based systems engineering that extends UML with executable semantics and supports model-driven approaches. OAL is part of xtUML and describes model behavior. With OAL, action definitions are platform-independent, facilitating the transformation from abstract models to platform-specific code [4-5].

Several studies have examined the transformation of dynamic models into source code. Among the notable ones is the article "Design of Rules for Transforming Sequence Diagrams into Java Code" [6], which presents work on transforming sequence diagrams into Java code using transformation rules and metamodels. This method specifies transformations to be applied when converting sequence diagram patterns into rule schemas for code generation.

Another significant publication is "Mapping UML Sequence Diagram into the Web Ontology Language" [7], which proposes a technique for transforming UML sequence diagrams into the Web Ontology Language (OWL). This work aims to develop a semantic web framework for UML models, not deviating from the primary UML model but utilizing the model for web-based system applications.

The study "Automatic Test Case Generation Using Sequence Diagram" provides a framework for generating test cases from UML sequence diagrams [8]. This proposed framework systematically extracts test cases, ensuring that the generated tests reflect the specified behavior of the sequence diagrams [9].

Furthermore, the article "Novel Approach to Transform UML Sequence Diagram to Activity" explains a unique methodology for transforming UML sequence diagrams into activity diagrams [10]. This approach focuses on preserving the behavioral semantics of the original sequence diagram while providing an alternative representation that may be more suitable for certain types of analysis and experiments. This methodology highlights the flexibility and adaptability of model transformations within MDD [11].

Therefore, transforming dynamic models into source code is a well-researched area within MDD. Our work further elaborates on existing approaches, considering the specifics of OAL and xtUML, and contributes to advancing model-driven techniques in the broader context of software engineering challenges.

3 Our Approach

In the following chapter, we describe the generation of source code from dynamic models, particularly sequence diagrams. The methodology consists of the following essential pillars and valuable tools: ANTLR for parsing and using the prototypes AnimArch and *SQD Tunder.*

3.1 Use of ANTLR

We began with ANTLR (Another Tool for Language Recognition) to parse the input JSON files that define the sequence diagrams. An example of the contents of these JSON files is shown in Figure 1. ANTLR is a powerful parser generator capable of interpreting, processing, executing, or translating structured text [12-13]. Figure 2 illustrates some of the rules of our parser grammar used by ANTLR for parsing the input files defining the sequence diagrams. However, we believed that ANTLR might not be as effective as we had anticipated due to the complexity of the input JSON files. The sequence diagrams were so complex that they required a more custom-tailored solution capable of handling the nuances of our data.

Error Handling and Edge Cases in Parsing

To ensure reliable operation and maintain output quality, our parsing process incorporates a multi-level error handling and recovery strategy:

- Lexical errors: Unknown or invalid characters are mapped to a special *error token*. The parser logs the error location (line, column) while continuing in *panic-mode recovery*, allowing other valid parts of the file to be processed.
- Syntactic errors: In *lenient mode*, we use ANTLR's DefaultErrorStrategy

- For local recovery (token insertion/deletion). In *strict mode*, we apply BailErrorStrategy to stop parsing at the first critical error.
- Semantic errors: After building the parse tree, we verify semantic correctness, including class and method existence, message sender/receiver validity and fragment structure consistency. Invalid references or malformed elements are reported with suggestions for correction.

Each incident is classified (LEX, SYN, or SEM) and recorded with its type, exact position in the source file, and recommended resolution. This approach allows us to process incomplete or partially erroneous inputs, generating OAL code from valid segments while marking unresolved elements for later revision.

3.2 Generating OAL Code

The generation of Object Action Language (OAL) code from sequence diagrams represented a core functionality of our method [14]. This process was modelled and documented in an activity diagram shown in Figure 3. The individual steps are outlined as follows:

1) **Creation of the Sequence Diagram:** The sequence diagram was created using the SQD Tunder prototype, which saves the diagram into a standalone file. This prototype is described in detail in Section 3.3.
2) **Analysis Using ANTLR:** We used ANTLR to perform lexical, syntactic, and semantic analysis. This process involves generating a lexer to split the input file into tokens, creating a parsing tree from these tokens, and traversing the parsing tree to collect information necessary for generating OAL code. Error handling during this stage is described in Section 3.1.
3) **Pre-processing:** Due to the complexity of the sequence diagrams, pre-processing was required. This step ensures that the parsed data is complete, consistent, and ready for translation into OAL code. Our pre-processing stage includes:
4) **Normalization:** Standardizing class names, lifeline identifiers, and message labels (case-insensitive comparison, removal of diacritics) to avoid duplicate or mismatched elements.
5) **Structural validation:** Verifying that control fragments (LOOP, ALT, OPT, PAR) are complete, properly nested, and syntactically correct. Unclosed or improperly nested fragments are flagged as errors.
6) **Reference validation:** Ensuring that every message has both a valid sender and a valid receiver, and that referenced classes or methods exist in the model.

7) **Default handling:** For optional attributes (e.g., missing guard conditions), safe default values are applied, and a comment is inserted in the generated code (e.g., // default guard: true).

8) **Detection of unused or isolated elements:** Identifying lifelines or messages not connected to the primary interaction flow. In *lenient mode*, these are retained but marked; in *strict mode*, they result in process termination.

9) **Error handling in pre-processing:**

   All detected issues are logged with their type, location, and recommended fix.

In *lenient mode*, non-critical errors (such as missing optional attributes or isolated elements) do not stop generation. These elements are either skipped or replaced with defaults, and the generated OAL includes // TODO comments.

In *strict mode*, any structural or reference error is considered critical and stops the generation process to ensure model integrity:

- **Translation to OAL Code:** The validated and pre-processed sequence diagram elements are translated into OAL code and stored in text files for subsequent execution and animation. Any unresolved elements are explicitly marked in the output code for later manual review.
- **Integration with AnimArch:** The generated OAL code is integrated into the AnimArch prototype for animation purposes. This integration assumes that the corresponding class diagram has been created and is consistent with the sequence diagram to ensure correct animation playback.

```
{
  {
      "XmiId":"MzczNzIyNjI0dWlsOkludGVyYWN0aW9uMDMvMjkvMjAyMyAxMDoyNTozMy44MTY=",
      "XmiType":"uml:Interaction",
      "lifeline":{
          {
              "XmiIdRef": "LTQwMzE3NjQ0OHVtbDpMaWZlbG1uZTAzLzI5LzIwMjMgMTA6MjU6MzMuODE2"
          },
          {
              "XmiIdRef": "MTY4OTAyNjU2dWlsOkxpvVsW1MDMwMjk0MjkwMjAyMyAxMDoyNTozMy44MTY="
          },
          {
              "XmiIdRef": "NzQwOTgxNzYwdWlsOkxpZmV51MDMvMjkvMjAyMyAxMDoyNTozMy44MTY="
          }
      },
      "message":{
          {
              "XmiIdRef":"MTMxMzA2MDg2NHVtbDpNZXNzYWdlMDMvMjkvMjAyMyAxMDoyNTozMy44MTY="
          },
          {
              "XmiIdRef":"MTg4NTEzOTk2OHVtbDpNZXNzYWdlMDMvMjkvMjAyMyAxMDoyNTozMy44MTc="
          },
          {
              "XmiIdRef":"LTE4Mzc3NDgyMjRlbWw6TWVzc2FnZTAzLzI5LzIwMjMgMTA6MjU6MzMuODE3"
          }
      },
      "fragment":{
          {
"XmiIdRef":"MzgwODM3MjQ4dWlsOk9jY3VycmVuY2VTcGVjaWZpY2F0aW9uMDMvMjkvMjAyMyAxMDoyNTozMy44MTY="
          },
          {
"XmiIdRef":"NDg3NTUlMzI4dWlsOk9jY3VycmVuY2VTCGVjaWZpY2F0aW9uMDMvMjkvMjAyMyAxMDoyNTozMy44MTc="
          },
          {
"XmiIdRef":"NTk0MjczNDA4dWls0k9jY3VycmVuY2VTcGVjaWZpY2F0aW9uMDMvMjkvMjAyMyAxMDoyNTozMy44MTc="
          },
          {
"XmiIdRef":"NzAwOTkxNDg4dWlsOk9jY3VycmVuY2VTcGVjaw2pY2F0aW9uMDMvMjkvMjAyMyAxMDoyNTozMy44MTc="
          },
          {
```


Figure 1
JSON files defining the sequence diagram

```
name
       : '"name"' ':' value
       ;

xmiId
       : '"XmiId"' ':' value
       ;

xmiIdRef
       : '"XmiIdRef"' ':' value
       ;

lifeline
       : STRING ':' '"uml:Lifeline"'
       ;

message
       : STRING ':' '"uml:Message"'
       ;

occurenceSpecification
       : STRING ':' '"uml:OccurrenceSpecification"'
       ;

combinedFragment
       : STRING ':' '"uml:CombinedFragment"'
       ;

interactionOperand
       : STRING ':' '"uml:InteractionOperand"'
       ;
```


Figure 2
Parser grammar rules

3.3 Prototypes: SQD Tunder and AnimArch

Two prototypes were developed to support the creation and animation of sequence diagrams: *SQD Tunder* and *AnimArch*:

SQD Tunder: This tool was used to create sequence diagrams, which were subsequently saved into standalone files for further processing. Figure 4 shows an example of this prototype.

AnimArch: Although AnimArch did not directly contribute to generating OAL code, it was extended to allow the creation of animations based on the generated sequence diagrams. These animations could be run in AnimArch, provided the corresponding class diagram had been created beforehand. Figure 5 illustrates the animated UML model in the AnimArch prototype. The separation between the SQD Tunder model creation interface and the code generation and animation logic in AnimArch follows a similar design principle to that of RAIN [15], where the REST API layer is decoupled from the asynchronous execution engine. Such separation improves responsiveness and enables concurrent processing without blocking user interaction.

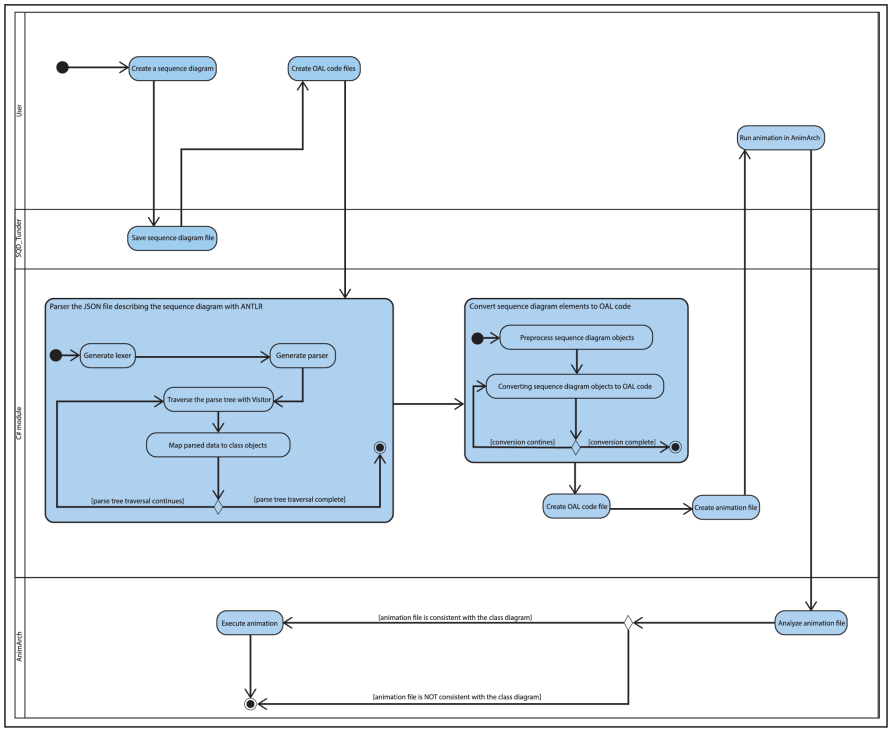

Figure 3

Process for Generating OAL Source Code from a UML Dynamic Model Sequence Diagram

3.4 Results

Our approach to transforming sequence diagrams into executable OAL code involved multiple phases of validation and refinement. The primary outcomes were as follows:

- **Transformation of sequence diagrams**: We successfully converted various sequence diagrams into OAL code, accurately capturing the dynamic behavior.
- **Validation of sequence diagrams:** Although validation was not a mandatory part of the translation process, sequence diagrams were checked for correct behavior. We confirmed their validity based on our predefined rules. In cases where the diagrams did not adhere to these rules, the user was notified of the discrepancies.

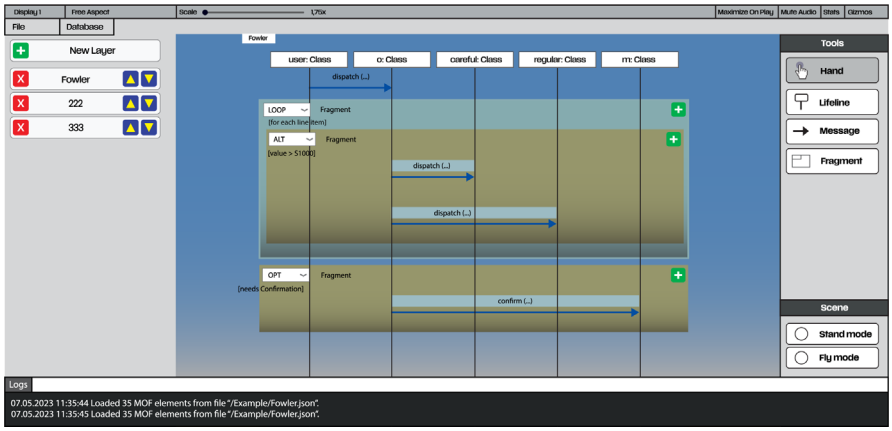

Figure 4
A sample of prototype *SQD_Tunder*

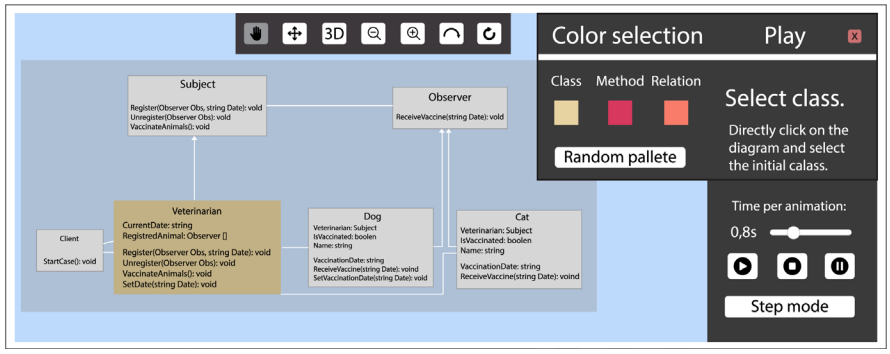

Figure 5
Example of running animation of UML model processes in the *AnimArch* prototype

We have successfully developed a robust and reliable process for generating code from dynamic models. The complexity of the input data presented significant challenges, which led us to adopt a cautious approach to iterative development. The following section presents a detailed evaluation and the results of our approach.

4 Evaluation

The generated source code in this study was evaluated by comparing the expected OAL code with the OAL code generated by our solution. This process required several steps, which we will present in the following subsection. Subsection 4.2 will show the sequence diagram cases upon which our solution is evaluated.

4.1 Evaluation Process

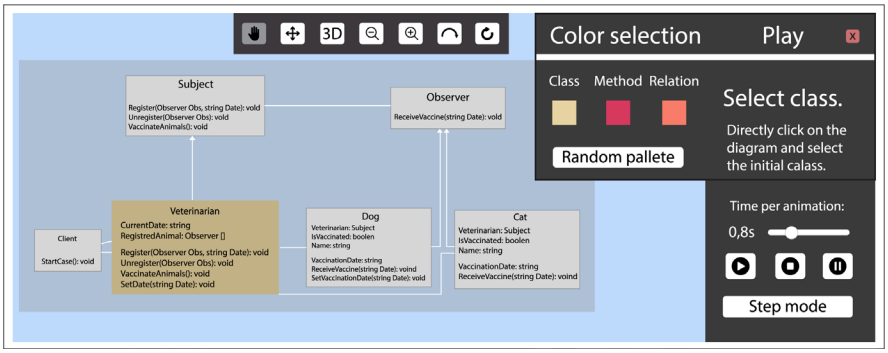

Figure 5

Example of running animation of UML model processes in the *AnimArch* prototype

This part presents our evaluation process, summarized in Figure 6. It follows from preparing the expected OAL code, generating and comparing the source code, and calculating the resulting precision and recall values.

Preparation of Expected OAL Code

We received functional OAL codes, from which we had to develop the most accurate sequence diagrams to describe them, or the functional OAL code was provided directly along with the sequence diagram. Using our method, we utilized the provided sequence inputs in each scenario to generate OAL codes. These OAL codes served as our benchmarks for relevance during the evaluation process.

Generation and Comparison

Through our implementation, we generated source code from the sequence diagram. We then compared the expected OAL code with the generated code by counting the specific elements. The elements considered included the number of classes, methods, instance creations, loops (for each and while), parallel blocks (par construct), and terminations of loops, commands, threads, and parallel blocks (end construct).

Comparison Methods

For comparison, we employed precision and recall metrics. False positives (FP) refer to elements in the generated OAL code that did not belong to the expected OAL code. In contrast, false negatives (FN) refer to elements missing in the generated OAL code but present in the expected OAL code.

Determination of Precision and Recall Values

We calculated precision and recall values based on the elements in the expected OAL and the generated OAL. Precision was determined as the ratio of correctly generated aspects to the total number of generated elements. Similarly, recall was calculated as the ratio of correctly generated aspects to the total elements inherent in the expected OAL.

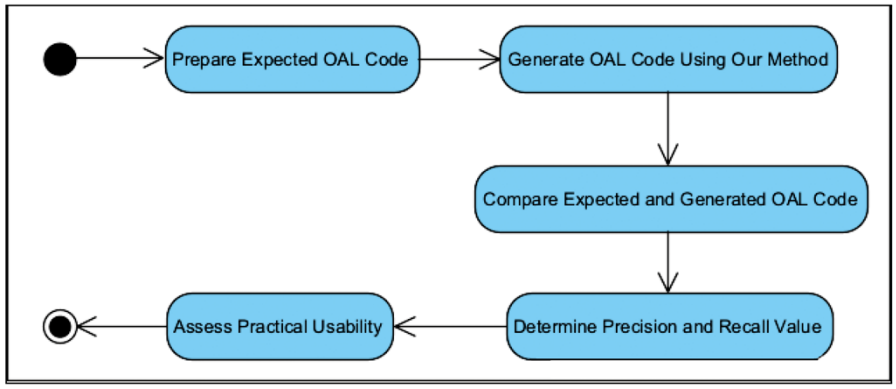

Figure 6
Activity diagram of the evaluation process

Practical Usability

We also validated the practical usability of the code generated by our method. This process involved evaluating the extent to which the generated code can be integrated and effectively utilized within the AnimArch prototype.

4.2 Specific Sequence Diagram Cases

Extended Evaluation Cases in response to the need for a more comprehensive evaluation, we extended our test set to include a broader range of UML sequence diagram scenarios reflecting real-world modelling challenges. In addition to the original cases, the following new cases were analysed:

- **Nested Parallel and Conditional Flows:** A complex diagram combining parallel execution paths (PAR) with nested conditional branches (ALT, OPT), testing the correct handling of concurrent and conditional interactions.
- **Error and Exception Handling:** A scenario representing message flows interrupted by error conditions and exception signals, assessing the system's capability to model and translate exceptional behaviour.
- **Multi-level Looping Structures:** Sequence diagrams with loops nested within loops, ensuring correct translation of repetitive interactions with varying loop conditions.
- **Integration with External Components:** A diagram demonstrating asynchronous communication between the primary system and an external service, validating message synchronisation and event handling.

For each case, precision and recall metrics were computed following the methodology described in Section 4.1. Results confirmed that the proposed approach maintained high accuracy across all scenarios, with only minor recall reductions in diagrams involving complex asynchronous flows. These results further validate the robustness of the method when applied to a broad spectrum of realistic modelling patterns.

The evaluation included six specific sequence diagram cases:

1. **Simple Sequence Diagram:** Assessed the basic continuous message flow
2. **Order of Messages:** Focused on the correct sequence of messages
3. **Sequence Diagram with Fragments:** Tested more complex interactions within the diagram
4. **Observer Pattern:** Evaluated using the observer design pattern
5. **Abstract Factory:** Evaluated using the abstract factory design pattern
6. **Parallel Processes:** Parallel processing within the abstract factory pattern was included

For instance, in the "Complex Interactions" evaluation case, the complex interactions within the sequence diagram were analyzed. The sequence diagram for this evaluation case (Figure 7) involves 4 classes: multiple messages distributed

throughout the diagram, a LOOP fragment, a nested ALT fragment within the LOOP fragment, and an OPT fragment. This diagram is from the book "UML Distilled: A Brief Guide to the Standard Object Modelling Language" by M. Fowler [16]. As with the other evaluation cases, the precision and recall metrics were calculated by comparing the generated and expected OAL codes. The comparison of OAL codes is illustrated in Figures 8 and 9.

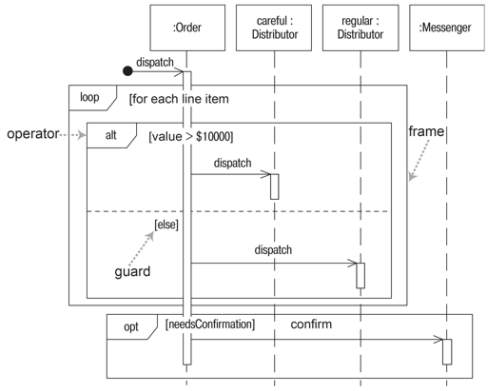

Figure 7
Sequence diagram used in the third evaluation case (Complex Interactions) [16]

```
class Order
    int[] items;
    bool needsConfirmation;
    method StartMethod()
        for each line_item in items
            if (line_item > 10000)
                create object instance inst_Careful_ of Careful;
                Careful_inst.dispatch();
            else
                create object instance inst_Regular of Regular;
                Regular_inst.dispatch();
            end if;
        end for;
        if (needsConfirmation)
            create object instance inst_Messenger of Messenger;
            Messenger_inst.confirm();
        end if;
    end method;
end class;

class Careful
    method dispatch()
    end method;
end class;

class Regular
    method dispatch()
    end method;
end class;

class Messenger
    method confirm()
    end method;
end class;
```

Figure 8
Sample of expected code for the 3rd evaluation case

```
class Order
    method StartMethod()
        for each line item
            if (value > 10000)
                create object instance Careful_inst of Careful;
                Careful_inst.dispatch();
            else
                create object instance Regular_inst of Regular;
                Regular_inst.dispatch();
            end if;
        end for;
        if (needsConfirmation)
            create object instance Messenger_inst of Messenger;
            Messenger_inst.confirm();
        end if;
    end method;
end class;

class Careful
    method dispatch()
    end method;
end class;

class Regular
    method dispatch()
    end method;
end class;

class Messenger
    method confirm()
    end method;
end class;
```

Figure 9
Sample of the generated code for the 3rd evaluation case

Table 1 below compares the number of elements in the generated and expected OAL codes.

As mentioned earlier, we calculated the precision and recall metrics to evaluate the accuracy of the generated OAL code. Precision was defined as the ratio of correctly generated OAL code elements to the total number of generated elements. Recall was defined as the ratio of correctly generated OAL code elements to the total number of elements in the expected OAL code.

In this evaluation, the value for True Positives (TP) was 21, for False Positives (FP) was 0, and for False Negatives (FN) was 2.

Table 1
Comparison of the number of code elements in the 3rd evaluation case

OAL Code Elements	Number of Elements in Expected Code	Number of Elements in Generated Code
Class	4	4
Method	4	4
Instance Creation	3	3
Method Call	3	3
for each Loop	1	1
if Statement	2	2
else Statement	1	1
end Construct	3	3
Class Attributes	2	0
All Elements	**23**	**21**

Using these values, the precision and recall were calculated as follows:

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{21}{21 + 0} = 1.0 \quad (1)$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{21}{21 + 2} = \frac{21}{23} \approx 0.91 \quad (2)$$

A high recall value indicates that the generated code contains no extraneous elements. In contrast, a slightly lower recall value suggests the absence of some elements compared to the expected OAL code. However, this gap does not imply the incompleteness of our approach; instead, it is a result of the limitations of the *SQD_Tunder* prototype in certain aspects. Specifically, the *SQD_Tunder* prototype does not support representing more complex structures, such as lists or assigning elements to variables. Therefore, the lower recall metric reflects the mentioned prototype's limitations, not our approach's shortcomings.

4.3 Overall Evaluation Results

The metrics analysis demonstrated that our developed method is applicable within the context of the AnimArch prototype. Comparison with the expected OAL code revealed only minor deviations in recall and precision metrics, indicating that the

approach can generate source code from sequence diagrams with a high degree of accuracy under the tested conditions.

The precision and recall results suggest that the method performs effectively for the evaluated cases, covering relevant elements identified and retrieved during the process. The expected OAL codes were used as relevance judgments to provide a consistent reference for comparison; however, we recognise that this measure reflects alignment with a predefined target output rather than serving as an absolute proof of system effectiveness.

While earlier evaluation was conducted on a single representative example, we have now extended the set of evaluation cases to include a broader variety of sequence diagram structures (see Section 4.2). This expansion provides a stronger basis for assessing the method's performance, though further large-scale testing on industrial-scale models would be needed to make definitive claims about robustness across all modelling scenarios.

The results indicate that the proposed approach is particularly suitable when the UML diagrams do not require code structures that are currently unsupported by the SQD Tunder prototype. For mapping more complex constructs into OAL code, the prototype would need to be extended in future work.

Conclusions

During this research, we developed and validated functionalities for generating source code from UML sequence diagrams in the OAL language for use within the AnimArch prototype. Our work represents a significant step forward for the future development of the AnimArch prototype [17], providing a robust and reliable foundation for further model-driven engineering research. The proposed method achieved high accuracy in transforming dynamic models into executable code, as confirmed by precision and recall analysis across a diverse set of evaluation cases.

The practical value of adaptive modelling and code generation approaches is further illustrated by related work such as *Scalable Real-Time Confusion Detection for Personalized Onboarding Guides* [18], which demonstrates the successful deployment of real-time user behaviour analysis in large-scale, multi-user environments. This alignment with real-world applications underscores the broader applicability of our approach within software engineering practice.

Future work will focus on extending the SQD Tunder prototype to support more complex code structures, such as lists, variable assignments, and advanced control flows, as well as on large-scale evaluation with industrial-grade models. These developments will further enhance the precision, coverage, and usability of the system, strengthening its role as a reliable tool for transforming UML sequence diagrams into executable OAL code within model-driven development workflows.

Acknowledgements

This work was supported by the following projects:

Funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under the project No. 09I05-03-V02-00055.

AI-Driven Self-awareness and Cognition for Compute Continuum, APVV-23-0430.

Decentralized artificial intelligence in a distributed virtualized computing environment 2/0081/26.
This work was carried out during the first author's PhD studies at the Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava.

References

[1] Pastor O, España S, Panach JI, Aquino N. Model-driven development. Informatik-Spektrum. 2008 Oct;31:394-407

[2] Selic B. The pragmatics of model-driven development. IEEE software. 2003 Sep 15;20(5):19-25

[3] Mellor SJ, Clark AN, Futagami T. Model-driven development. IEEE software. 2003 Sep 1;20(5):14

[4] Mellor SJ, Balcer MJ. Executable UML: a foundation for model-driven architecture. Addison-Wesley Professional; 2002

[5] xtUML.: Action Language (OAL) Tutorial. [Accessed 3-November-2023]. Avail-able at: https://xtuml.org/learn/action-language-tutorial/

[6] Thongmak M, Muenchaisri P. Design of rules for transforming uml sequence diagrams into java code. InNinth Asia-Pacific Software Engineering Conference, 2002, 2002 Dec. 4 (pp. 485-494) IEEE

[7] Elkashef N, Hassan YF. Mapping UML sequence diagram into the web ontology language OWL. International Journal of Advanced Computer Science and Applications. 2020;11(5)

[8] Panthi V, Mohapatra DP. Automatic test case generation using sequence diagram. In Proceedings of International Conference on Advances in Computing 2012 (pp. 277-284) Springer India

[9] Tatale S, Chandra Prakash V. Combinatorial test case generation from sequence diagram using optimization algorithms. International Journal of System Assurance Engineering and Management. 2022 Mar;13(Suppl 1):642-57

[10] Kulkarni DR, Srinivasa CK. Novel approach to transform UML Sequence diagram to Activity diagram. Journal of University of Shanghai for Science and Technology. 2021;23(07):1247-55

[11] Vahdati A, Ramsin R. Modeling and Model Transformation as a Service: Towards an Agile Approach to Model-Driven Development. In International

Conference on Lean and Agile Software Development 2022 Jan 12 (pp. 116-135) Cham: Springer International Publishing

[12] Parr T. The definitive ANTLR 4 reference

[13] Tomassetti G. The ANTLR mega tutorial. Federico Tomassetti-Software Architect (2017-03-08) [2020-08-19] https://tomassetti. me/antlr-mega-tutorial 2017

[14] Ciccozzi F, Malavolta I, Selic B. Execution of UML models: a systematic review of research and practice. Software & Systems Modeling. 2019 Jun 1;18:2313-60

[15] Habala, Ondrej, Martin Šeleng, Michal Habala, Ľubor Stuhl, Michal Staňo, and Ladislav Hluchý. "Scalable Cloud Application Deployment Service for Versatile Cloud Service Deployment and Configuration." Computing and Informatics 43, No. 6 (2024): 1416-1431

[16] Fowler M. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional; 2018 Aug 30

17] Radosky, Lukas, and Ivan Polasek. "Executable multi-layered software models." In Proceedings of the 1st International Workshop on Designing Software, pp. 46-51, 2024

[18] Hucko, Michal, Robert Moro, and Maria Bielikova. "Scalable Real-Time Confusion Detection for Personalized Onboarding Guides." In International Conference on Web Engineering, pp. 261-276, Cham: Springer International Publishing, 2020