Comparing LLM-based and MDE-based code generation for agile MDE

Qiaomu Xue[1], Kevin Lano[1]

[1]King's College London, London, UK

Abstract

Large Language Models (LLMs) are increasingly applied to many different software tasks, and by means of specific domain training, they can perform well in specialised areas. Code-aware LLMs can generate executable program code from natural language specifications and prompts, but there are questions about the reliability of this process, because the results are not always consistent and correct. Model-driven engineering (MDE) also provides code generation techniques, which produce code automatically based on software designs expressed in UML or OCL. Compared to the LLM solution, the code generation process in MDE is more explicit, and the transformed code is determined by the source model, and there is usually a high correctness level for such generation. The code-generation by example (CGBE) process uses symbolic machine learning to synthesise code generators for particular code generation processes in MDE. The resulting generators map from UML/OCL specifications to the target programming language. In this paper we compare the effectiveness of code generation using baseline and fine-tuned LLMs, such as DeepSeek and LLama2, with CGBE code generators.

Keywords

Code generation, Model-driven engineering, Machine learning, Large language model

1. Introduction

Software systems are increasingly used in many different application areas, and hence the needs for software programming skills and development expertise have risen to a high level in many industrial sectors. Model-driven engineering (MDE) is one solution to increase software production, and it relies on using models as artefacts to underpin the software engineering development process and support automated software production. In the MDE paradigm, automated software production by transformation from design models to executable code avoids errors caused by manual programming, and narrows the gap between the design and the final product. In addition, it can generate highly reliable and accurate results, and improve code traceability and verifiability. In an agile MDE process, automated code generation provides rapid system updates in response to changing requirements.

The process of MDE includes building models of software specifications and designs, and defining a transformation (code generator) that maps the source models to target executable code [1]. Customised code generators may need to be produced for new MDE applications, so to support an agile MDE process, a means of rapidly producing accurate code generators is necessary. MDE requires high levels of skills in practitioners, who must have knowledge of modelling, and knowledge of the metamodels and semantics of the source and target languages involved. Manually constructing a code generation transformation from UML and OCL to a programming language such as Java, C or Python is difficult because of the scale and complexity of the languages, requiring highly skilled developers and substantial development time. Moreover, each code generator in MDE is usually specific to only one target programming language, so the construction effort needs to be repeated for different targets, and the process is costly in time and resources. Therefore, a method for building code generators that requires lower skills and that can construct generators to multiple target programming languages is important for agile MDE. Our project for this goal is *code generation by example* (CGBE) which supports building a code generator using

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10–13, 2025.

qiaomu.1.xue@kcl.ac.uk (Q. Xue); kevin.lano@kcl.ac.uk (K. Lano)
0000-0002-9706-1410 (K. Lano)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-4122/paper13.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2025-12-10

symbolic machine learning (ML) [2]. This approach can address the problem of producing accurate code generators for multiple target programming languages, as well as reducing the human effort and time.

Large language models (LLMs) [3] also provide a code generation solution that can deliver result code for natural language requirements given via a conversational interface. Following recent refinements and improvements in LLM capabilities for code generation, LLMs are now used and accepted by developers for practical programming tasks. LLM tools focused on code generation include Copilot, DeepSeek and Cursor. However, using LLMs for code generation does not always meet the demand for accuracy and consistency, because they have variable and stochastic behaviour, and their outcomes cannot be completely controlled. LLMs may sometimes generate syntactically-correct code that is not correct in its function [4]. Also, the black-box nature of LLMs prevents developers from knowing how the LLM makes coding decisions [5]. Considering the potential advantages and disadvantages of MDE and LLMs approaches for code generation, we want to precisely compare the CGBE code generation approach of MDE with LLM solutions, applied to the same problems (generation of Kotlin and JavaScript code from UML and OCL). The comparison will consider code generation accuracy, the skills needed for application, and construction effort for the different approaches.

2. Related Work

Here we review recent related work in code generation using MDE and LLMs.

2.1. Recent MDE work in code generation

MOBICAT [6] is a model-driven method that supports GUI development of Android apps, emphasising domain-specific modelling in order to streamline mobile development. They improve the automated MDE process for GUI generation but still encounter scalability problems and platform-specific issues, which are general problems of MDE. Likewise, the related mobile app approach of AppCraft [7, 8] is limited to code synthesis for problems which can be expressed in its specific domain-specific language (DSL) for ML-enabled mobile apps.

Except in unusual cases, MDE is now often combined with other technologies to achieve functionality. The paper [9] applies MDE to machine learning engineering, supporting ML system formalisation, reuse and maintenance, but they encounter problems of high cost on small tasks and high complexity on large tasks as well as poor traceability. In [10], the authors consider the potential of LLMs to improve the automation of modelling, addressing the difficulty of implementing complex design problems, as well as easing the process of coding, but the disadvantages are reduced accuracy and reliability. The use of LLMs in an MDE process may demand extra human validations or checking because of the risk of hallucinations of LLMs (plausible but incorrect outputs). These problems are also considered by [11]. In their research LLMs are used to facilitate model-driven engineering by generating code from models, in order to address the problem of lack of semantic depth in traditional modelling, that is, the models omit details of functional behaviour, which the LLM can complete. They find that the use of LLMs improves agility, although uncontrolled errors may arise in the end code, and still have to be checked by human beings. The survey [12] points out that MDE is mature in the language engineering aspect for supporting AI systems software engineering but lacks an understanding of business processes and collaborations. They consider that the combination of MDE and AI is still at an initial stage. While AI has huge potential to enhance and improve MDE processes, it still has limited adoption in the domain. In [13] the authors apply deep neural networks to generate or modify code directly from models. This has similar drawbacks to LLM-based code generation, with challenges in achieving bug-free code, security and reliability issues, and data bias in training datasets may lead to biased or unfair code generation.

2.2. Recent LLM work in code generation

Large Language Models (LLMs) are seen as a good solution for addressing text-based problems. Code-based tasks such as code summarisation, refactoring, translation and generation are addressed by code-aware LLMs. Coding advice can be provided both by general LLMs such as GPT-4, and by specialised LLM-based tools like Copilot, which provide code completion and other support for developers to assist programming tasks. The survey [14] examines LLM performance in code generation, they confirm that there have been advancements in LLM capabilities, and emphasise the quality of datasets having a key impact on LLMs ability for code generation. The paper [15] provides a benchmark for increasing the performance of machine learning for code generation and programming understanding tasks. They conclude that there remains accuracy problems and limitations in ML understanding of programming language semantics. The study [16] identifies that LLMs perform much better at generating individual statements and methods, rather than complete semantically-coherent classes. The survey [17] advises that LLMs need to have increased understanding of code semantics to enable LLMs to integrate smoothly into the development workflow.

3. Methodology

In this section we describe the compared code generation approaches.

3.1. Code generation by example (CGBE)

CGBE is a code generation solution utilising a symbolic machine learning method inspired by inductive logic programming (ILP) [2]. It is a specialised form of model transformation by example (MTBE), and learns code generation transformation mappings from corresponding examples of the source and target languages, and it expresses these mappings as explicit transformation rules in the CSTL text-to-text transformation language. CGBE is integrated into the AgileUML toolset [18]. Figure 1 shows the CGBE process for code generator construction.

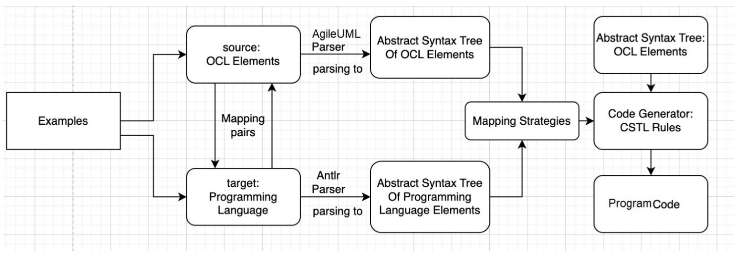

Examples
source: OCL Elements
target: Programming Language
Mapping pairs
AgileUML Parser
parsing to
Antlr Parser
parsing to
Abstract Syntax Tree Of OCL Elements
Abstract Syntax Tree Of Programming Language Elements
Mapping Strategies
Abstract Syntax Tree: OCL Elements
Code Generator: CSTL Rules
ProgramCode

Figure 1: CGBE process

Typically, the source language for CGBE is UML/OCL and the target programming languages are procedural/object-oriented languages. CGBE has been applied to synthesise code generators for Kotlin, JavaScript and Visual Basic. The textual source and target language examples are parsed into abstract syntax trees (ASTs), and tree-to-tree structural mappings are discovered between the sets of trees, using various search strategies. These include tree-to-sequence, sequence-to-sequence and tree-to-sequence strategies.

To carry out code generator synthesis using CGBE, sufficient training examples need to be prepared, covering all the UML/OCL constructs. The examples are usually grouped into four categories: 1. types; 2. expressions; 3. declarations (including use cases and classes, with attributes and operations); 4. statements in the AgileUML operational OCL dialect. For each example, the target programming language elements that correspond to the source element should be defined. For instance, in the Kotlin case, if the source element is the *boolean* type in AgileUML OCL, then the corresponding target element is the *Boolean* Kotlin type. The outcome of the CGBE process is a set of mapping rules in CSTL format, which are grouped into grammar-based categories. The CSTL transformation can then be validated, and if it is sufficiently correct (usually we require 100% correctness) it can be utilised as a plugin for AgileUML to generate code in the target language for general UML/OCL specifications.

An example set of CSTL rules created by CGBE for the JavaScript code generator are:

```
OclStatement::
var _2 : double   |-->let _2 = 0.0 ;
var _2 : boolean   |-->let _2 = false ;
var _2 : String   |-->let _2 = "" ;
var _2 : Sequence   |-->let _2 = [] ;
var _2 : _4  |-->let _2 = 0 ; <when> _4 Integer
while _2 do _4  |-->while ( _2 ) { _4 }
_1 := _3  |-->_1 = _3 ;
if _2 then _4 else _6  |-->if ( _2 ) { _4 } else { _6 }
for _2 : _4 do _6  |-->for ( var _2 of _4 ) { _6 }
return _2  |-->return _2 ;
break  |-->break ;
continue  |-->continue ;
return  |-->return ;
```

These define the mapping of OCL statements to JavaScript program statements.

3.2. Large language models (LLMs)

LLMs are large scale pre-trained ML models which can provide a general capability for text analysis and generation. They are often utilised via a conversational interface whereby the LLM replies to user questions. General LLMs can be specialised for specific tasks by further training on domain-specific data to apply in specialised areas, e.g., for program translation [19]. To achieve high LLM performance in a specialised task, some fine-tuning method involving supervised learning is usually applied [3]. We will use the fine-tuning approach to retrain a selected LLM to become expert at MDE code generation. LLama2 is chosen by us as the LLM to fine-tune.

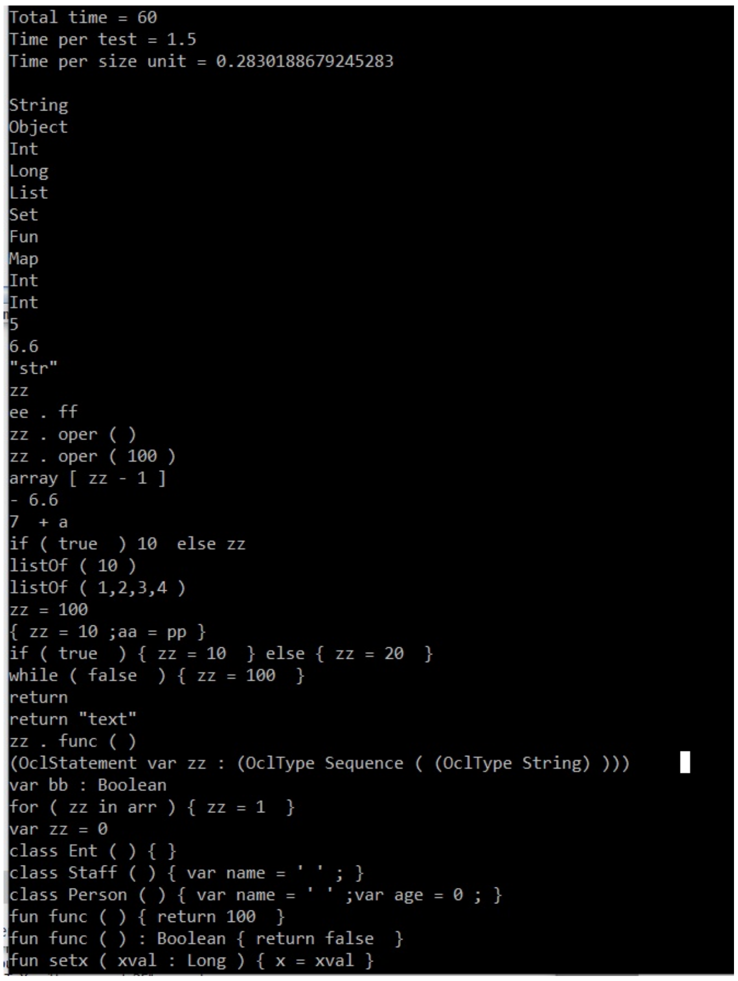
In some cases, an LLM could be used directly for code generation. In particular, DeepSeek is a new LLM that is already recognised for its high reasoning abilities and logical coding capabilities. Therefore we will directly use the DeepSeek LLM via its API to address the same task of code generation from UML/OCL to the target programming language for the experiment. DeepSeek-R1 is chosen for our experiment.

4. Experiments and Evaluation

In this section we will compare the performance of the three code generation approaches in terms of their accuracy, the level of user skills needed, and the effort required to build the solutions. Two code generation tasks are considered: UML/OCL to Kotlin, and UML/OCL to JavaScript. These targets are languages of different types (Kotlin is a strongly-typed object-oriented language, whilst JavaScript is a weakly-typed scripting language with a prototype-based semantics). We will apply CGBE, a fine-tuned LLM and direct LLM invocation for these tasks and compare their results. For each target language, we use a common dataset of examples to train or prompt each code-generation method. Evaluation data is at: `zenodo.org/records/15387656`.

4.1. Kotlin case with CGBE

To apply code generation by example, we define a training dataset consisting of four groups of examples concerning types, statements, expressions and declarations. In these groups, the examples are presented in pairs, normally the left-hand side (LHS) is a UML/OCL element, and the right side (RHS) is the corresponding target language element(s). For CGBE, the left-hand sides are normally fixed examples of UML/OCL and are the same for all code-generation tasks, but the right-hand side elements need to be filled by corresponding target language elements. For the Kotlin case the total number of examples are 189 pairs. To use CGBE for Kotlin code generator synthesis, a Kotlin parser should be installed, the parser maps the RHSs of the training pairs to abstract syntax tree (AST) format according to the Kotlin grammar. The LHS of pairs are parsed into AST format by the AgileUML parser for UML/OCL. CSTL rules are then automatically derived by the search-based process of CGBE, which looks for structural mappings from the ASTs of the example LHSs to the ASTs of the example RHSs. A validation dataset disjoint from the training dataset is also prepared for validating the transformation ability. There are 40 validation cases to check the generation of Kotlin code, and only one OCL case did not transform correctly (Figure 2). The incorrect result is identified by the presence of an untransformed OCL AST in the results, this indicates that no applicable rule was found in the CSTL script to generate target code from the AST, i.e., from an OCL declaration statement `var zz : Sequence(String)`. In turn, this error indicates incompleteness in the training data and hence that this should be extended and the CGBE procedure iterated.



```
Total time = 60
Time per test = 1.5
Time per size unit = 0.2830188679245283

String
Object
Int
Long
List
Set
Fun
Map
Int
Int
5
6.6
"str"
zz
ee . ff
zz . oper ( )
zz . oper ( 100 )
array [ zz - 1 ]
- 6.6
7  + a
if ( true  ) 10  else zz
listOf ( 10 )
listOf ( 1,2,3,4 )
zz = 100
{ zz = 10 ;aa = pp }
if ( true  ) { zz = 10  } else { zz = 20  }
while ( false  ) { zz = 100  }
return
return "text"
zz . func ( )
(OclStatement var zz : (OclType Sequence ( (OclType String) )))
var bb : Boolean
for ( zz in arr ) { zz = 1  }
var zz = 0
class Ent ( ) { }
class Staff ( ) { var name = ' ' ; }
class Person ( ) { var name = ' ' ;var age = 0 ; }
fun func ( ) { return 100  }
fun func ( ) : Boolean { return false  }
fun setx ( xval : Long ) { x = xval }
```

Figure 2: CGBE validation results for Kotlin

4.2. Kotlin case: Fine-tuning using LLama2-7b

Based on the available computing resources and recent information on code-aware LLMs, we chose LLama2-7b as the baseline LLM. We aim to build a fine-tuned LLM from this baseline, which can be used to transform UML/OCL to Kotlin code. The initial training dataset is the CGBE Kotlin examples in the paired training format (Section 4.1). We applied fine-tuning three times with different sized training datasets based on this core dataset. We tested and evaluated these three different fine-tuned LLM versions on the transformation task.

4.2.1. LLM with zero-shot learning

We first applied the LLama2-7b LLM without fine-tuning or other background information.

When applied to simple cases of OCL expressions and statements the results were poor, with mostly irrelevant responses. These tests for the original LLama 2-7b model show that without additional training or contextual information it has poor accuracy for code generation, and may produce incorrect and irrelevant responses.

4.2.2. First iteration of fine-tuning

The CGBE examples dataset was divided into a training dataset, a test dataset and a validation dataset. Throughout the fine-tuning process, we used the validation dataset to measure the accuracy of the trained model, however the scores of the results were extremely low. We also used the chat dialog to check the performance of the trained model by conversation with it. The machine translation accuracy indicators BLEU-4 and ROUGE-1 were also used for measuring the accuracy, however these are general textual accuracy measures, and code has additional (semantic) correctness concerns compared to natural language, hence a double check through conversational inquiry is necessary. Out of 28 validation questions for OCL to Kotlin transformation, only 8 had correct replies.

4.2.3. Second iteration of fine-tuning

Because of the low scores in the first iteration of fine-tuning, perhaps due to the limited training data, we attempted a second iteration of fine-tuning with increased training dataset size. The new data retained the same format but we augmented the original data by producing new variants with differently-named variables. For instance, an example with OCL text $1/p$ would be repeated as $1/q$ and $1/r$. The enlarged dataset has 489 samples. We also changed the parameters of fine tuning depending on the evaluation results, and achieved improved results with the loss rate decreased from 1.77 in the first iteration to 0.112 in the second. The accuracy scores using the validation dataset improved, with BLEU-4 = 72.3318, ROUGE-1 = 94.2955, ROUGE-2 = 84.7446, ROUGE-l = 93.3355. For the 28 test cases there are 17 correct answers from this fine-tuned version.

4.2.4. Third iteration of fine tuning

We further expanded the training data using data augmentation to more than 1000 samples. However, the resulting accuracy scores are similar to the second iteration results, and the results of the validation dataset are also similar. Therefore the performance of fine-tuning did not improve by further increasing the dataset samples.

4.3. Kotlin case: Direct LLM API invocation

The fine-tuning method strongly relies on data for training. In contrast, directly invoking an LLM API does not require construction of training data. In this test the DeepSeek-R1-8b API is invoked. This model was selected in order to be of comparable size to the LLama2 model used for fine-tuning.

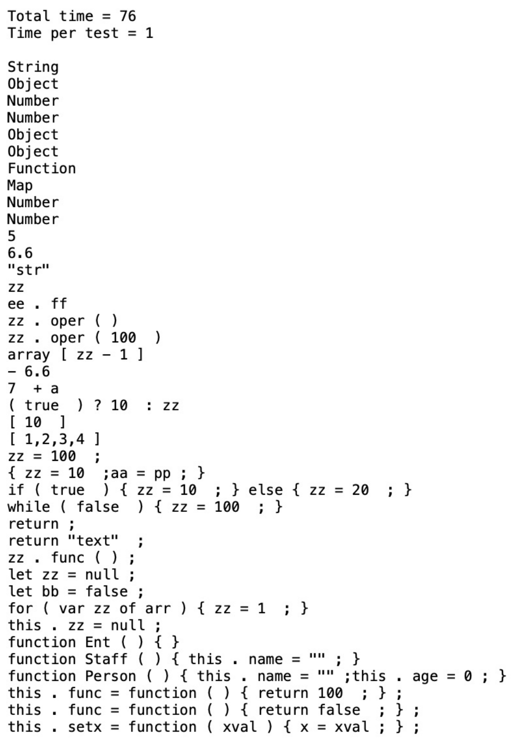
The data comes from the example pairs of CGBE used in Section 4.1. We select only the LHSs of these examples, with the Kotlin part deleted: we expect this part to be generated by the LLM. The data is put in one file and the examples are input to the LLM one item at a time. The prompt is:

```
Convert the following OCL expression into Kotlin code.
Output only Kotlin code, without explanations.
```

For the 182 input data items the LLM produces 38 syntactically-invalid answers according to the Kotlin grammar, and there are a further 22 answers with semantic problems. Extraneous reasoning content is sometimes produced, that should not be generated based on the prompt.

4.4. JavaScript case with CGBE

For JavaScript, there are 165 examples used as training input for CGBE. After learning the rules of the code generator, the same OCL test cases used for Kotlin are executed for validation. Figure 3 shows the results of the CGBE performance in JavaScript case. All 40 validation examples are translated correctly in this case.



```
Total time = 76
Time per test = 1

String
Object
Number
Number
Object
Object
Function
Map
Number
Number
5
6.6
"str"
zz
ee . ff
zz . oper ( )
zz . oper ( 100  )
array [ zz - 1 ]
- 6.6
7  + a
( true  ) ? 10  : zz
[ 10  ]
[ 1,2,3,4 ]
zz = 100  ;
{ zz = 10  ;aa = pp ; }
if ( true  ) { zz = 10  ; } else { zz = 20  ; }
while ( false  ) { zz = 100  ; }
return ;
return "text"  ;
zz . func ( ) ;
let zz = null ;
let bb = false ;
for ( var zz of arr ) { zz = 1  ; }
this . zz = null ;
function Ent ( ) { }
function Staff ( ) { this . name = "" ; }
function Person ( ) { this . name = "" ;this . age = 0 ; }
this . func = function ( ) { return 100  ; } ;
this . func = function ( ) { return false  ; } ;
this . setx = function ( xval ) { x = xval ; } ;
```

Figure 3: CGBE validation results for JavaScript

4.5. JavaScript case: Fine-tuning using LLama2-7b

The same 165 examples were used for fine-tuning of LLama2 for JavaScript code generation. As with Kotlin, this small dataset led to poor results with BLEU-4 = 0.1914, ROUGE-1 = 5.1625, ROUGE-2 = 0.307, ROUGE-l=0.953. For 10 validation tests, all 10 results were incorrect. After increasing the dataset size to 532 samples using further examples, the training loss was reduced to 1.0596, and the indicators of prediction are BLEU-4 = 54.5213, ROUGE-1 = 79.0119, ROUGE-2 = 59.3292, ROUGE-l = 73.701. Validation results gave an accuracy of 40%. After further increasing the dataset size to 999 samples, loss was further reduced to 0.1681, with indicators BLEU-4 = 83.8775, ROUGE-1 = 94.5479, ROUGE-2 = 87.6406, ROUGE-l = 92.3393. 90% of validation examples were processed with syntactically correct results and 80% were semantically correct. In contrast to the Kotlin case, we manually prepared additional examples to produce the larger datasets, rather than simply copying examples and renaming variables.

4.6. JavaScript case: Direct LLM API invocation

We applied the DeepSeek-r1-7b LLM to each of the left-hand side OCL elements of the training dataset. For every case the prompt was "Convert the following OCL expression into JavaScript code. Output only JavaScript code, without explanations.". The results were syntactically incorrect in 27% of cases, that is, the generated code was invalid in the JavaScript grammar. There were further cases with semantic errors due to the LLM misunderstanding OCL elements.

4.7. Discussion

We applied three approaches to solve the code generation task in MDE, with one MDE-based approach and two LLM-based, each applied to both Kotlin and JavaScript code generation tasks. We can compare the results by measures of code quality, effort needed for construction, and the requirements needed for application of the approach.

4.7.1. Code quality

Regarding code quality there are several aspects to assess. For accuracy, the produced code should at least follow the grammar of the target programming language, and additionally it should correctly express the source semantics. Table 1 summarises the results for the three approaches applied to Kotlin and JavaScript, where for Kotlin we select the second iteration of fine-tuning, and for JavaScript the third iteration.

Table 1
Accuracy of code generation approaches

Approach	*Kotlin case correctness*		*JavaScript case correctness*	
	Syntactic	*Semantic*	*Syntactic*	*Semantic*
CGBE	97.5%	97.5%	100%	100%
Fine-tuned LLM	60.7%	53.5%	90%	80%
Direct API LLM	79%	67%	73%	62.5%

For Kotlin CGBE validation only one of 40 cases was incorrectly generated, but the generation process can guarantee 100% accuracy in principle if there is sufficient source language coverage in the dataset. For JavaScript 100% accuracy was achieved.

For Kotlin, the best performance of the fine-tuning method is the second iteration of fine-tuning, which produced 17 syntactically correct answers in 28 test cases, however the semantic accuracy is not high. The direct API approach has 38 syntactically incorrect outputs from the 182 samples tested. It ensures basic accuracy of the transformation (79% syntactic correctness).

For JavaScript the direct API approach was unable to produce JavaScript for some OCL elements, and its performance was somewhat lower than for Kotlin. However, fine-tuning produced good results. The difference to the Kotlin case may be that the expanded JavaScript dataset has improved diversity and distribution of data, resulting from manual construction of new examples, rather than by automated data augmentation of the base dataset.

The scalability of the three approaches is also likely to be different, since generators produced by CGBE apply the CSTL rules mechanically to their input, regardless of the input size and complexity, whereas LLM code generation tends to deteriorate with increasing input size [16].

4.8. Development effort

For CGBE the developer must provide manual translations for the training dataset, which is relatively small for both target language cases (around 200 examples), but which may need to be extended for particular target languages to guarantee the accuracy of the generated code. This can involve iteration of the training and validation testing procedures.

The fine-tuning method relies on the choice of training parameters and training data to ensure high accuracy. More data and experimentation with changing parameters are needed for building higher accuracy code generators. Again, this is an iterative procedure which may need several rounds to achieve satisfactory results. Typically larger datasets are needed than for CGBE, and producing large high quality datasets may require significant manual effort.

The accuracy of the direct LLM API invocation approach depends upon the chosen model and the prompts defined by the developer. It may be that more detailed prompts can lead to better accuracy, however sometimes an LLM does not follow the prompts, for instance, our prompts to DeepSeek asked for no explanation, yet the output still had explanations. The direct API answers are not uniform in format and code style, for example some are only expressions and some are extended to a function.

4.9. Application requirements

For CGBE, the developer needs to understand the UML/OCL and target language syntax and semantics. They also need to install the parser (such as an ANTLR parser) of the target language to run the CGBE function. For the fine-tuning approach the developer needs to understand the source and target language syntax and semantics, and the management of the LLM fine-tuning process. Directly invoking an LLM API is the simplest approach for code generation, because the developer only needs to know how to invoke the model and give it correct prompts. For the convenience of code generation, direct invocation of LLMs is the preferable way to get quick replies for a batch of requests.

4.10. Summary

In the accuracy aspect the CGBE approach has the best performance, and is the only approach which can achieve reliable and repeatable 100% accuracy. In the application requirements aspect, the direct API invocation is the simplest, because there is no construction needed, and only low user skills.

Conclusions

In this paper, we compared three approaches for code generator construction, and applied them to solve the same problems of transforming UML/OCL specifications to Kotlin and to JavaScript. We investigated the advantages and disadvantages of the approaches in terms of generated code quality, the construction process and the requirements needed for application of the approach. The three methods have their own characteristics and appropriate application scenarios, but for code generation where accuracy is the first priority, an MDE approach such as CGBE appears to be the most suitable choice. The disadvantages of CGBE in generating code may be reduced if it is possible to combine it with AI techniques to provide user support to reduce the skills needed (such as dataset construction) to apply the approach.

Due to resource constraints, we were limited to the comparison of relatively small LLMs. In future work we will compare CGBE with larger-scale LLMs such as DeepSeek-r1-70B or Llama3, and also consider a wider range of code-aware LLMs, such as StarCoder, GPT-4 and Mistral.

Declaration on Generative AI

The authors used DeepSeek and LLama2 to generate code examples, in order to compare the results to MDE code generation. All outputs were reviewed and validated by the authors, who take full responsibility. No proprietary/third-party confidential data were provided to these tools.

References

[1] K. Lano, S. Yassipour-Tehrani, H. Alfraihi, S. Kolahdouz-Rahimi, Translating from UML-RSDS OCL to ANSI C, in: OCL 2017, STAF 2017, 2017, pp. 317–330.

[2] K. Lano, Q. Xue, Code generation by example using symbolic machine learning, Springer Nature Computer Science (2023).

[3] W. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, et al., A survey of large language models, arXiv 2303.18223v10 (2023).

[4] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, LLMs for software engineering: a systematic literature review, arXiv 2308.10620 (2023).

[5] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, J. Zhang, Large language models for software engineering: Survey and open problems, in: ICSE '23, 2023.

[6] H. Zafar, S. Khan, A. Mashkoor, H. Nisa, MOBICAT: a model-driven engineering approach for automatic GUI code generation for Android applications, Frontiers in Computer Science 6 (2024).
[7] L. Alwakeel, K. Lano, H. Alfraihi, Towards integrating machine learning models into mobile apps using AppCraft, in: AgileMDE workshop, STAF 2023, 2023.
[8] L. Alwakeel, K. Lano, H. Alfraihi, AppCraft: Model-driven development framework for mobile applications, IEEE Access (2025).
[9] S. Raedler, M. Rupp, E. Rigger, S. Rinderle-Ma, Code generation for machine learning using MDE and SysML, arXiv (2023).
[10] J. D. Rocco, D. D. Ruscio, C. D. Sipio, P. Nguyen, R. Rubei, On the use of large language models in model-driven engineering, SoSyM (2025).
[11] A. Sadik, S. Brulin, M. Olhofer, A. Ceravola, F. Joublin, LLM as a code generator in agile model-driven development, arXiv (2024).
[12] S. Raedler, L. Berardinelli, K. Winter, A. Rahimi, S. Rinderle-Ma, Bridging MDE and AI: a systematic review of domain-specific languages and model-driven practices in AI systems engineering, SoSyM (2024).
[13] J. Wang, H. Xu, C. Xiao, Y. Zheng, X. Wu, Semantic research on model-driven code generation, in: EMIE 2024, 2024.
[14] J. Jiang, F. Wang, J. Shen, S. Kim, S. Kim, A survey on large language models for code generation, arXiv (2024).
[15] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, CodeXGLUE: a machine learning benchmark dataset for code understanding and generation, arXiv (2021).
[16] X. Du, M. Liu, K. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, Y. Lou, Evaluating large language models in class-level code generation, in: ICSE 2024, 2024.
[17] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, J. Chen, A survey of llms for code, arXiv (2024).
[18] Eclipse Agile UML project, 2025. Projects.eclipse.org/projects/modeling.agileuml, accessed 8.1.2025.
[19] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, V. Ganesh, CoTran: An LLM-based code translator using reinforcement learning with feedback from compiler and symbolic execution, arXiv:2306.06755v4 (2024).