From Natural Language to Web Applications: Using Large Language Models for Model-Driven Software Engineering

Lukas Netz[1], Judith Michael [2], Bernhard Rumpe [3]

Abstract: We evaluate the usage of Large Language Models (LLMs) to transform natural language into models of a predefined domain-specific language within the context of model-driven software engineering. In this work we test systematically the reliability and correctness of the developed tooling, to ensure its usability in an automated model-driven engineering context. Up to now, LLMs such as ChatGPT were not sophisticated enough to yield promising results. The new API-Access and the release of GPT-4, enabled us to develop improved tooling that can be evaluated systematically. This paper introduces an approach that can produce a running web application based on simple informal specifications, that is provided by a domain expert with no prior knowledge of any DSL. We extended our toolchain to include ChatGPT and provided the AI with additional DSL-specific contexts in order to receive models that can be further processed. We performed tests to ensure the semantic and syntactic correctness of the created models. This approach shows the potential of LLMs to successfully bridge the gap between domain experts and developers and discusses its current limitations.

Keywords: ChatGPT, GPT-4, Model-Driven Engineering, Code Generation, MontiGem, DSL

1 Introduction

Domain-Specific Languages (DSLs) aim to reduce the gap between the domain expert and the Software Engineer [FR07, FL10] by raising the level of abstraction, while at the same time reducing the design space to match the needs of the targeted problem domain [KT08]. The usability of any DSL is a key factor in its acceptance by the domain expert. However, there are many challenges in designing a highly usable DSL [Ba14]. Among the challenges the domain expert has to learn when creating a new model in a new DSL are: (1) syntax and semantics of the language, (2) how to compose the syntax to perform a function, (3) comprehension of syntax written by others, (4) debugging of syntax, and (5) modification of a model.

With the increasing sophistication of Large Language Models (LLMs) [Ch21, Li22] such as ChatGPT [Br20, Op23b], the question arises as to whether these can be used to perform the model definition for the domain expert, similar to [Ar16] and [Sa20b]. The domain

[1] RWTH Aachen University, Software Engineering, Ahornstraße 55, 52074 Aachen, Germany netz@se-rwth.de

[2] RWTH Aachen University, Software Engineering, Ahornstraße 55, 52074 Aachen, Germany michael@se-rwth.de

[3] RWTH Aachen University, Software Engineering, Ahornstraße 55, 52074 Aachen, Germany rumpe@se-rwth.de

expert only would have to specify the desired use case, leaving the LLM to generate the semantically and syntactically correct model. Within this work, we want to evaluate *to what extent generative pre-trained transformers can be used, to reliably produce data structure models of a given DSL.* We limit ourselves to data structure models in this approach, as they serve as input for a generator framework (MontiGem [Bu24, Ge20b]), which transforms class diagrams into a running web application.

In order to narrow down our evaluation, we will proceed under the following assumptions:

1. The input is provided in natural language as informal requirements by a domain expert (e.g. a subject matter expert on finance or healthcare).

2. The domain expert might have no prior knowledge about the used DSL, thus he might not be able to validate any response directly from the LLM.

3. The domain expert can validate if a resulting artifact, such as a generated application, based on the GPT output, fits his requirements or not.

As we rely on multiple language models, we have to take several perils into account, that come with using a LLM-based system [Bo21, Ja23]: (1) **Non-determinism** poses a challenge when depending on a tool to produce valid models consistently. The language model may generate varied output for the same request, resulting in instances where it provides a textual model at one time and only the model description at another. Even when fine-tuning responses with a low-temperature setting, the outcomes remain unpredictable. (2) **correctness** of the output is questionable, as it can not be guaranteed that the training data the language model is trained on is correct. Data might be outdated, not present, or simply wrong, resulting in similarly faulty output by an LLM. Further aspects to discuss are (3) the amount to which the **choice of the target DSLs** impacts the likeliness to synthesize a valid model, (4) if an approach to transform informal specifications into a DSL is feasible, compared to **alternatives** such as the direct transformation into a GPL (e.g. using ChatGPT to create Java code directly), and (5) **limitations** that are introduced by using LLMs.

In this work, we explore the effectiveness of Language Models (LLMs) in producing data structure models for web application development. To achieve this, we introduce a transformer that takes informal specifications from a domain expert as input and constructs a prompt. This prompt is then fed to an LLM, which in turn produces a model. The toolchain can subsequently utilize this model to generate a web application.

The paper is structured as follows: We introduce fundamentals such as the tools and DSLs we use in section 2. We follow up with a description of the proposed approach (section 3), before listing the results in section 4, which are discussed in section 5. After presenting the related work in section 6, the last section concludes.

2 Used Technology and Languages

The DSL CD4A presented in this work is developed with MontiCore. MontiCore [KRV08, HKR21] is a language workbench for the efficient development of DSLs. Language engineers define a context-free grammar to describe the abstract syntax of the language (i.e. what constructs are valid models in the DSL). The concrete syntax has a textual representation.

Context Conditions (CoCos) are used to further specify the model, as the context-free grammar does not support such conditions. Based on these artifacts, MontiCore generates mainly three infrastructures: A parser for models, a traverse/visitor infrastructure, and a symbol table. The parser generates an abstract syntax tree (AST) based on the information in the model. During this process, the input models' consistency with the grammar is also validated. The resulting AST can then be traversed using the provided traverser/visitor infrastructure. The symbol table is used to find symbols in the model and in the scope of the model. This is useful in order to check whether a reference in one model to another model is valid and whether the referenced symbol exists, thus providing the means to easily validate many contextual conditions. This infrastructure supports the language developer to create a DSL and generate code from its models. Within this work, we use the generated infrastructure (especially the parser and cocos) to evaluate any models that are synthesized for a MontiCore-based DSL.

```
1  classdiagram CD {
2    class Person {
3      String name;
4      Date birthday;
5    }
6    class Student extends Person{
7      long studentId;
8    }
9    class Animal {
10     String name;
11   }
12   association [1]Person -> Animal[*];
13 }
```

List. 1: CD4A Class Diagram Defining Person, Student and Animal Class and their relations.

Class Diagrams for Analysis (CD4A) is a textual DSL developed with MontiCore. It is based on UML [Ob17] to define class diagrams (CDs) in a Java-like syntax [Ru16]. In addition to attribute- and class definitions, CD4A supports all common elements of CDs, e.g., *associations*, *inheritance*, and *enumerations* (see [Ch23]).

Listing 1 shows an example of a class diagram consisting of the class Person with two attributes name and age. The class Student inherits from Person. Additionally, there is an association between Person and Animal, modeling optional Pets for each Person.

2.1 MontiCore-Based Generator for Enterprise Management (MontiGem)

MontiGem [Ge20b, Ad18, Ge20a, Mi22] is a generator-framework based on the MontiCore language workbench. It meets enterprise information system needs, which are inherently data-centric [JWM06]. It uses models from UML/P [Ob17] as input artifacts, such as class diagrams and OCL as well as GUI models to generate the target code in a server-client

architecture (similar to [HMMM18]). The code generation mainly handles the creation of boilerplate code and provides basic infrastructure, allowing the developers to focus on the business logic. The configuration of MontiGem used in this work only requires a class diagram, to generate the web application. An internal model-to-model transformation (CD2GUI) derives GUI models for each data class defined in the input class diagram, leaving the definition of GUI models and OCL constraints optional.

The framework has been used in a variety of real-world projects and teaching activities. MontiGem has been extended to generate, e.g., low-code development platforms for digital twins [Da22], process-aware digital twin cockpits [Ba22], assistive systems [MRZ21], or IoT App stores [Bu22].
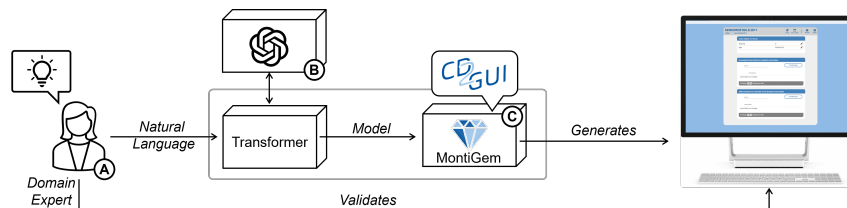
3 Methodology

Fig. 1: Domain-Expert defining an application without knowledge of any DSL used.

The following experimental setup (c.f.Figure 1) is used to systematically evaluate both versions of the LLM (GPT-3.5 and GPT-4). A domain expert (A) states informal requirements as written input to the transformer. The transformer uses a configuration that is tailored to the targeted DSL and uses LLM, to provide the language model (B) both with the user request as well as additional instructions on how to transform it into the correct DSL. The transformer extracts and validates the model from the response with a corresponding parser and either requests a new one or provides the model to a generator that can process the model further. We use (C) MontiGem as a generator for the web application. The running application can easily be evaluated by the domain expert, as only domain knowledge (instead of expertise in modeling) is needed for the evaluation. Note that instead of MontiGem other generator frameworks could be used, e.g., M3D [Be12]. Once the application is created, the user inspects the outcome and if needed has the option to try again or to adapt the model with natural language. As we intend to run a large number of tests and the generation and inspection of an entire application is relatively time-consuming, we focus on the creation and validation of models. We proceed under the assumption that MontiGem (C) produces an application for every valid CD4A model it is provided [Mi22].

The prompts we transmit to the LLM must exhibit a high level of fine-tuning to obtain answers that fulfill two key criteria: a) accurate formatting and syntax (such as including a textual model), and b) relevance to the specific use case addressed by the question. In the following example (compare with Figure 2), we use an API to instruct the Language Model

to generate a Class Diagram. However, while the response is semantically accurate, it bears no resemblance to the specified DSL CD4A, rendering it unsuitable for further processing within a CD4A-based toolchain.

User: *Create a CD4A model for a Ping Pong game.*

GPT-4: *Sure! Here's a simple CD4A class diagram for a Ping Pong game:*

```
1  +---------------+
2  | Ball          |
3  +---------------+
4  | + move()      |
5  | + bounce()    |
6  +---------------+
7
8  +---------------+
9  | Paddle        |
10 +---------------+
11 | + moveUp()    |
12 | + moveDown()  |
13 +---------------+
14
15 Paddle *-- Ball
```

Is there anything else you would like to know?

User: *Please generate a Monticore CD4A model describing a Ping Pong game*

GPT-4: *How does the model start?*

User: *A Monticore CD4A model starts with* `classdiagram Name {` *where* `'Name'` *should be the name of the model.*

GPT-4: *Understood. How are public and private variables declared?*

User: *Public and private variables are declared using the* `'public'` *and* `'private'` *keywords, respectively, rather than with* `'+'` *or* `'-'` *symbols.*

Fig. 2: An unconfigured interaction with GPT, can yield CDs in the wrong syntax.

Fig. 3: Providing additional specifications and context is necessary to yield parsable results.

As GPT-3.5 [Br20] and GPT-4 [Br20] are LLMs that consider the context of the current conversation, we can use the API to provide a fictive conversation that already established key rules of the CD4A grammar (c.f. Figure 3).

By iterating on the key elements of the target DSL, we can increase the chance of getting the correct response in the correct syntax. Note that we do not need to explain every rule or production of the corresponding DSL grammar to get a valid answer. Using the API provided by OpenAI [Op23a], we can provide a fictive predefined conversation to LLM with both versions of the LLM *before* asking any domain-specific questions, thus priming the language model to more likely return a valid model (c.f. Figure 3). We can reuse previous conversations that yielded a high rate of valid models or alternatively compose a set of prompts in which we define the correct information, in order to create a model with the correct syntax. Similar success can be reached by providing examples of the given DSL. Few-Shot Learning (FSL) [Br20, Oh22] distinguishes between three training modes. (1) Few-Shot: The LLM is given a task with a few demonstrations of possible solutions for the task. (2) One-Shot: The LLM is provided with a single demonstration. (3) Zero-Shot: No demonstration is given, the answer only relies on the pre-trained data of the LLM. As FSL

purely relies on the context provided with the task, we can provide it in a similar fashion as with prompts without the need to costly retrain the LLM.

The toolchain (c.f. Figure 1) is configured to set up a conversation in which we pass on the informal specifications from a domain expert to get a correct textual model as a response. The model is used for subsequent generator steps, creating an information system. Providing LLM with a predefined context improves the consistency of the response: As the system is configured to be non-deterministic, there is a high chance that follow-up questions by LLM would differ from previous iterations resulting in a slightly different setup of the LLM each time. Providing context increases the chance of getting a model with the correct syntax. Since a parser is part of the tools we get generated by MontiCore for each DSL, we can use it to systematically evaluate the success rate of the LLMs responses.

4   Evaluation

To evaluate the success rate of the LLM's responses, we have evaluated it with the following tasks and several cases:

1. **Creating a CD4A model:** Following the setup shown in Figure 1, we will iteratively produce models and count the rate of valid models.

2. **Creating a PlantUML Model:** We evaluate the performance again.

3. **Evaluating semantic correctness:** We provide LLM with a task from an exam and assess the results based on a grading schema.

4. **Adapting existing models:** We evaluate the success rate of modification of a textual model, based on a natural language input.

4.1   Creation of a CD4A model

To assess the reliability of generating a semantically meaningful and syntactically correct model, we conducted multiple iterations of the same query (results can be

User: *Please generate a Monticore CD4A model of a simple user interface.*

Predefined context: *A Monticore CD4A model starts with 'classdiagram' Name { where 'Name' should be the name of the model.* [...]

GPT-4: [...][a]

```
1   classdiagram SimpleUserInterface {
2     class Button {
3       public char symbol;
4     }
5     class Label {
6       public char text;
7     }
8     class TextField {
9       private int maxLength;
10      public char inputText;
11    }
12    class Window {
13      public Button button;
14      public Label label;
15      public TextField textField;
16      public void openWindow();
17      public void closeWindow();
18    }
19  }
```

---
[a] Additional 'chat'-text is omitted to increase readability.

Fig. 4: Creating a CD4A model with GPT-4

found in Table 1). It's important to note that the use cases were intentionally underspecified, allowing the Language Model to generate 'creative' solutions. When tasked with creating a Class Diagram for a basic user interface using GPT-4 (as shown in Table 2), the models generated exhibited significant variability in terms of both size (ranging from 60 to 20 lines of code) and the type of graphical user interface (GUI) modeled, spanning from a collection of GUI elements to user management or data validation interfaces. The average cosine similarity of all models is 0.52. One example run is shown in Figure 4. GPT-4 delivers almost always a parsable model. In a few cases in which models could not be parsed by the CD4A parser contained the symbol `ArrayList`. The LLM failed to define this additional type and therefore produced an invalid Model. To reduce the systematic error based on specific use cases, we ran multiple use cases (Table 1).

Both GPT-3.5 and GPT-4 have a high chance of delivering a valid CD4A model. The LLM GPT-4 performs better than its predecessor, as it uses a more sophisticated language model [Op23b]. The success rate does not seem to be affected by the choice of use case, as long as they are comparable in their complexity (See a use case with higher complexity in subsection 4.3).

OpenAI notes that it might perform worse on knowledge on which it has little or no training [Op23b]. It has a high chance of providing a correct model for a generic product but could return semantically incorrect models of a specific product with a specific configuration. To test this, we provide the LLMs with a narrow target domain:

> **User:** *Please generate a data structure model of Lego Set 75192.*

Out of 20 requests, 19 provided a valid model. Only 4 described the specific Lego set stated (Millennium Falcon). The remaining models described generic Lego sets.

Model	Use Case	Valid	N	Rate
GPT-3.5	Ping Pong game	15	20	75%
GPT-4	Ping Pong game	99	100	99%
GPT-3.5	Simple User Interfaces	76	100	76%
GPT-4	Simple User Interfaces	97	100	97%
GPT-4	Digital Twin of a Hydraulic Press	19	20	95%
GPT-4	Lego Set 75192	19	20	95%

Tab. 1: Success rate of different use cases evaluated over N iterations

Our initial tests provided the LLM with instructions on how to write a CD4A model. According to [Br20, Oh22] we should be able to reach similar results by providing examples instead of instructions. As shown in Table 2, both LLMs are very unlikely to produce correct syntax if not given any examples. The likelihood increases for both GPT-3.5 and GPT-4 if an example is given. However, GPT-3.5 returned fewer valid models when given more than one example. The performance of GPT-4 improved with further models.

The LLM is very unlikely to produce the correct syntax if only provided with the task to transform the specification to a specific DSLs, assuming that the DSL is not widely used

Language Model	Kind of Context	Valid	N	Rate	Similarity
GPT-3.5	Zero-Shot	0	100	0%	0.08
GPT-3.5	One-Shot	51	100	51%	0.20
GPT-3.5	Few-Shot	26	100	26%	0.22
GPT-4	Zero-Shot	0	20	0%	0.12
GPT-4	One-Shot	73	100	73%	0.25
GPT-4	Few-Shot	88	100	88%	0.32

Tab. 2: Success rate of the same task (Creating a CD4A model) using different contexts, evaluated over N iterations

and, thus, the LLM is less likely to be trained on that DSL. We can teach the LLM by either giving examples of the DSL (FSL) or by providing it with a few instructions on how to adhere to the syntax. We have achieved a success rate of up to 99% for CD4A models using GPT-4 (Table 1), despite the lack of training data on the internet for this specific DSL. CD4A is tailored for developers and closely resembles the Java syntax. We hypothesize that this success rate can be attributed to the language model's training on Java code and UML models. This familiarity with Java syntax likely simplifies the generation of code with a similar structure, contributing to the high success rate observed.

4.2 Creating a PlantUML Class Diagrams

Another DSL that defines UML Class Diagrams in PlantUML. Assuming that GPT-4 is already trained on Class Diagrams, we expect similarly high success rates as for CD4A. The PlantUML language covers multiple diagram formats, e.g., CDs, state charts, activity diagrams, BPMN, and component diagrams. As described in section 3, we use a parser to validate the syntactic correctness of any returned model.

Model	Use Case	Valid	N	Rate	Similarity
GPT-4	Ping Pong game	20	20	100%	0.33
GPT-4	Simple User Interface	18	20	90%	0.24
GPT-4	Exam Task	18	20	90%	0.63

Tab. 3: Success rate of different use cases evaluated of N iterations. Cosine Similarity of models increases with more restrictive use-case descriptions.

As we can see in Table 3, the change in DSL does not have much effect on the success rate of the approach. We reach success rates (90%-100%) that are comparable to the ones measured with CD4A.

4.3 Evaluating Semantic Correctness

Semantic correctness of a data structure model is hard to measure [Ve20], and often a point of discussion. In the following, we will provide the LLM with a task from an exam. We

can use the grading schema of the exam to evaluate the produced models and compare the performance of the transformer with the results of students who took the same exam. The assessment for this task has a maximum score of 13.5 points. The grading criteria assess the diagram's correctness, considering various elements like compositions, inheritance, association cardinalities, and common mistakes in model design. The task reads as follows:

> **User:** *Generate a class diagram of an EBike according to these specifications:*
> *The EBike is composed of a frame (made out of steel), a drive system, and a controller. Two wheels are inserted into each frame. The drive system is composed of a motor. Each EBike can be connected to a removable battery. The battery has a stored energy measured in Watt-hours (Wh). The controller can be in one of three states: On, Off, and Charging. It also controls the battery, if one is connected, and commands the drive system. The company plans two different variants of the controller, a basic controller, and an advanced controller. The advanced controller should be able to estimate the next Date the bike should be inspected for maintenance.*
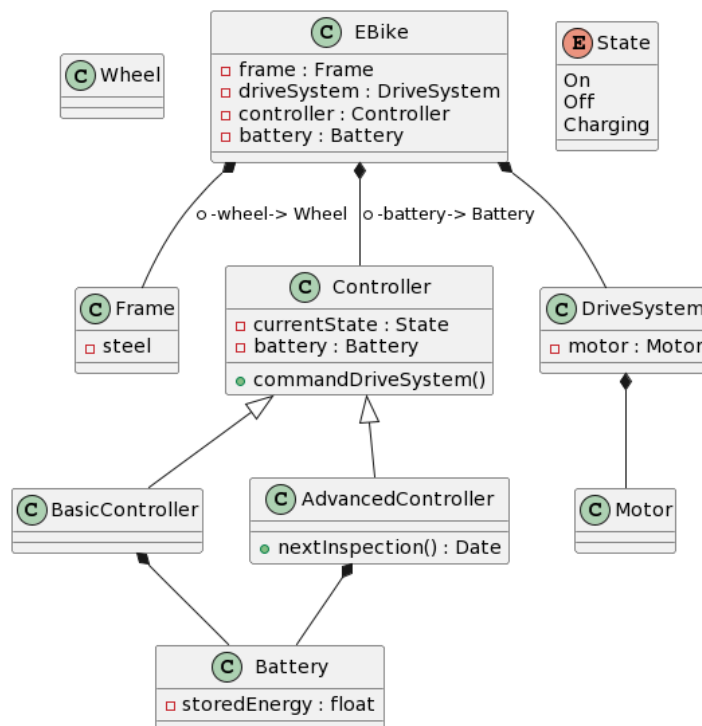

Fig. 5: Visualization of the Textual Model extracted from a GTP-4 response. This model was graded with 8.5/13.5 Points according to the grading schema of the exam. Some semantic errors are a missing relation from `Frame` to `Wheel` and a wrong data type for `storedEnergy`

We graded 20 models produced by GPT-4. On average 7.5/13.5 points were reached. The lowest score was 5 points, and the highest was 10.5. Three models failed the task as they scored below 50%. A group of 40 students who had written the exam scored an average of 10.0 points. This task is an entry-level assignment for the students, thus almost all students scored more than 50%. Nevertheless, it shows that the LLM is likely to produce accurate models for precise requirements.

4.4 Adapting Existing Models

We must assess the feasibility of extending and altering specific aspects of a model, based on the informal specifications of a domain expert. Thus far, our evaluation has centered on the efficiency and reliability of an LLM-based approach in generating new models. Our next step is to examine whether this approach can facilitate iterative editing of existing models.

We test as follows: We use a CD from one of our research projects containing about 30 classes (for a model excerpt see [Mi19]) as input. Next, we instruct our tool to add a specific class. Additionally, we define a new context, setting up theLLM to extend the model in the correct syntax.

Out of 20 iterations, we succeeded with 19 correct PlantUML models, each containing a user class. Using CD4A, we succeeded with all 20 models.

An interesting side effect we noted was the adaptation of model-specific properties by the LLM. Within the model, we used the uncommon `ZonedDateTime` type to define time-based attributes. In cases where the LLM also modeled a time-based attribute, it would adopt this type (e.g. `dateOfBirth`). Similar effects could be seen with commenting styles (c.f. Figure 6): In cases where the input CD is divided into sections that are marked with large comment blocks (e.g. `/*******` ...), the LLM showed a tendency to reuse this style, but customized it to its change: The introduction of commentary into the model can prove to be a problem when iterating over the same model multiple times. A refinement of the provided context to ban the LLM from adding comments could reduce this problem.

> **User:** *Add a user class and an association between the user class and* `Person` *to this model:* [...]

GPT-4: [...]

```
[...]
  /*******************************
   * New class and association
  *******************************/
  class User {
    String name;
    String email;
    String role;
  }

  Association User <-> Person [*];
[...]
```

Fig. 6: GPT-4 adapting provided CD4A models

5   Discussion

We have shown, that it is possible to synthesize models of a given DSL via a LLM and discuss the challenges and limitations of this approach.

Problems with non-determinsim. Almost all problems with non-determinism arise from underspecification of the task provided to the transformer. By configuring the API to reuse the same context for each request, and by restricting the LLM to a specific target-DSL, we consistently get positive results. An indicator of the impact of non-determinism is the average difference between all generated models measured with cosine similarity. By providing a more restrictive use-case description we can reduce this impact as shown in Table 3. A more precise task will lead to the generation of models that are more similar to each other. Note that we used the same temperature configuration of 0.8 for the API throughout all measurements.

Problems with correctness. As stated in the introduction, a common peril with LLM is the uncertainty of getting a correct answer. In our case, we only consider the validity of the model syntax and leave the semantic correctness to be checked by the domain expert. We are able to check every response with a parser removing any uncertainty about the syntax of the returned model. Regarding semantic correctness, we ran several tests showing that LLM is very likely to provide a fitting response, but we are unable to guarantee a correct model for every iteration. Further research has to be done to validate semantic correctness.

Problems with unknown DSLs and generalizability. We tested this approach with two DSLs describing a rather common UML-Diagram type: the class diagram. Some of the success of this approach is certainly due to LLM being trained on the concepts of UML and the availability of lots of class diagram models on the internet to train from. It is to be expected that models other DSLs such as UI-description languages are harder to produce, although we also noted that LLM is able to adapt on given input (subsection 4.4) and can be supported with a fitting context. This needs to be tested further.

Using multiple chained generative approaches. Existing solutions, like CoPilot [BJP22] and Codex [SVD21], can generate source code directly from natural language input. However, it's important to note that these solutions, as of the time of writing, are not suitable alternatives to the approach described here. Their primary focus is on assisting developers in extending pre-existing code rather than generating complete applications. In addition to using a model as an artifact to systematically define software, this approach also provides a level of consistency a language model cannot provide due to its non-determinism, and limited context size. Producing several thousand lines of code is very unlikely to be reproducible, in contrast to reproducing two very similar models as shown in subsection 4.3. Direct code generation is yet very limited and is ideally used for smaller coding tasks for example in education [Fi22].

Limitations of this approach. Although we try to reduce the required knowledge of the domain expert to a minimum, we still can not prevent faulty usage of the tool. As we pass on

the requirements defined by the domain expert, we can not validate the user input and, thus, can not guarantee correct results. E.g., we assume that the user does not specify the syntax. We will most likely get bad results, should the user provide input asking for a different DSL than the one the tool is currently configured for. As this tooling is in its early stages, we will look for mechanisms to simplify its usage.

Secondly, we are limited by the current context size of the used LLM. The current version of the ChatGPT-API only supports a context consisting of up to 8.000 tokens. The class diagram used in subsection 4.4 consisted of 30 classes and is represented by 1.647 tokens. Another larger class diagram consisting of 100 classes would be represented by 12.031 tokens and could not be processed. Future releases will support 32.000 tokens leaving more room for larger models.

Finally, the approach is limited by the training data the LLM was trained on. Information that is newer than 2021 is yet unknown to the AI. Similarly, it will not be able to provide information on data that is not publicly available. It is most likely that the training data will be updated at some point, but as of the time of writing, we are working with the data set as of 2021. OpenAI announced the introduction of plugins, allowing ChatGPT to access current data from the internet.

Up to now, we see three main threads to validity:

A) *Using a Beta version of ChatGPT:* ChatGPT is constantly under development. Thus, it might yield different results in future iterations. The access we used were both beta and early access versions that are likely to change over time.

B) *Choice of context and DSLs:* Another thread is the choice of a predefined context for each used DSL and the choice of the DSLs itself. A context was chosen that yielded promising results. Further evaluation might yield a better-suited context configuration that improves upon the results shown above. We tested the approach with multiple DSLs to mitigate DSL specific biases.

C) *Non-Determinism:* Due to the non-determinism of ChatGPT results vary over multiple iterations. To reduce this effect, we ran tests at least 20 and up to 100 times.

6 Related work

The concept of utilizing natural language for generating code or models has been investigated in several studies, highlighting its potential in bridging the gap between domain experts and developers. Fill, Fettke and Köpke perform in [FFK23] a series of experiments in which ChatGPT is used to synthesize UML Models. Rather than focusing on statistics of one type of model, their work gives a broad overview of the options as a modeling tool that ChatGPT can provide to a modeler. Desai et al. [De16] proposed a general framework for creating synthesizers that translate natural languages into specific DSLs. Although the approach

is narrow and relies on available training data, the synthesizers exhibit high accuracy in their translations. Ibrahim and Ahmad [IA10] developed a method for extracting CDs from textual requirements by employing natural language processing techniques and domain ontology. The Requirements Analysis and CD Extraction (RACE) tool aims to streamline the requirements analysis process, improving the efficiency of software engineers. Pang et al. [Pa20] introduced a method for converting UI sketches or images into DSL code and subsequently into executable code using an attention-based deep neural network, similar to GPT. They proposed two novel models, HGui2Code and SGui2Code, to address various challenges. HGui2Code focuses on the meaning and context of the GUI and DSL code, while SGui2Code emphasizes adherence to the DSL grammar. Interestingly, the mixed attention model demonstrated higher accuracy. Ernst [Er17] delved into the notion of using natural language as a programming language. By demonstrating the code generation capabilities of an AI model, he shows the potential to bridge the gap between domain experts and developers. Thomas et al. [Th22] conducted a comprehensive survey on programming with natural language. The study traces the development of natural language as a programming language. Early approaches that worked with context-free grammars culminated in more modern approaches using large language models such as GPT. This progression underscores the increasing role of natural language in the field of software engineering and its potential for bridging the communication gap between domain experts and developers. The presented approach could be seen as a low-code or no-code approach. Low-code development platforms target similar challenges to enable domain experts to create applications with little to no expertise in software development [Sa20a, RR16].

Future Work. As the LLMs are still under development, the performance of the approach is likely to improve. Further improvements to the input modes of the API are planned by OpenAI such as the support of image and document processing. An extension of the approach to also support this input mode could be followed up on. Another interesting approach would be to construct sets of models that relate to each other. For example, a CD that is constrained by a set of OCL models. Finally, as MontiCore grammars themselves are also defined in a DSL, it would be interesting to evaluate the creation of new grammars with LLMs. A language engineer could provide the LLM with a set of requirements, and use it to create a corresponding DSL.

7   Conclusion

We were able to demonstrate a methodology that uses LLMs to produce semantically correct and syntactically valid models with a high success rate. Once created, models can be passed directly to other frameworks. In our case, a generator framework is used to create a complete web application. We also could show that this approach is not limited to one specific DSL and can be applied to others.

GPT-4 delivers a higher performance throughout all tested DSLs. We often reached better results using custom prompts instructing GPT to key aspects of the target DSL, using up

less context and, thus, being more cost-efficient and leaving more room for more complex models. We also showed that few-shot learning is a valid and more systematic approach to teaching an LLM on how to transform natural language into a DSL. However, factors such as overfitting the provided models have to be taken into account.

Although there are still some issues with the presented approach, the results are very promising. One key issue, the validation of semantic correctness, remains to be done by hand. Although all syntactic valid models seem to be semantically valid as well, *the final check has still to be performed by a domain expert.* Enabling the domain expert directly to define domain-specific models with informal descriptions, improves the usability of the model-driven approach tremendously. It *gives the domain expert more time to focus on the requirements regarding his domain*, than investing time and effort into figuring out the intricacies of the used DSL-syntax.

Bibliography

[Ad18]   Adam, Kai; Netz, Lukas; Varga, Simon; Michael, Judith; Rumpe, Bernhard; Heuser, Patricia; Letmathe, Peter: Model-Based Generation of Enterprise Information Systems. In: Enterprise Modeling and Information Systems Architectures (EMISA'18). volume 2097 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 75–79, 2018.

[Ar16]   Arora, Chetan; Sabetzadeh, Mehrdad; Briand, Lionel; Zimmer, Frank: Extracting Domain Models from Natural-Language Requirements: Approach and Industrial Evaluation. In: ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems. MODELS '16, ACM, NY, USA, p. 250–260, 2016.

[Ba14]   Barišic, Ankica; Amaral, Vasco; Goulão, Miguel; Barroca, Bruno: Evaluating the usability of domain-specific languages. In: Software Design and Development: Concepts, Methodologies, Tools, and Applications, pp. 2120–2141. IGI Global, 2014.

[Ba22]   Bano, Dorina; Michael, Judith; Rumpe, Bernhard; Varga, Simon; Weske, Matthias: Process-Aware Digital Twin Cockpit Synthesis from Event Logs. Journal of Computer Languages (COLA), 70, June 2022.

[Be12]   Bernardi, Mario Luca; Cimitile, Marta; Di Lucca, Giuseppe Antonio; Maggi, Fabrizio M: M3D: a tool for the model driven development of web applications. In: 12th Int. WS on Web information and data management. pp. 73–80, 2012.

[BJP22]   Barke, Shraddha; James, Michael B; Polikarpova, Nadia: Grounded copilot: How programmers interact with code-generating models. arXiv preprint arXiv:2206.15000, 2022.

[Bo21]   Bommasani, Rishi; Hudson, Drew A; Adeli, Ehsan; Altman, Russ; Arora, Simran; von Arx, Sydney; Bernstein, Michael S; Bohg, Jeannette; Bosselut, Antoine; Brunskill, Emma et al.: On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258, 2021.

[Br20]   Brown, Tom; Mann, Benjamin; Ryder, Nick; Subbiah, Melanie; Kaplan, Jared D; Dhariwal, Prafulla; Neelakantan, Arvind; Shyam, Pranav; Sastry, Girish; Askell, Amanda et al.: Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

[Bu22]   Butting, Arvid; Kirchhof, Jörg Christian; Kleiss, Anno; Michael, Judith; Orlov, Radoslav;
           Rumpe, Bernhard: Model-Driven IoT App Stores: Deploying Customizable Software
           Products to Heterogeneous Devices. In: ACM SIGPLAN Int. Conf. on Generative
           Programming: Concepts and Experiences (GPCE 22). ACM, pp. 108–121, 2022.

[Bu24]   Buschhaus, Constantin; Gerasimov, Arkadii; Kirchhof, Jörg Christian; Michael, Judith;
           Netz, Lukas; Rumpe, Bernhard; Stüber, Sebastian: Lessons Learned from Applying
           Model-Driven Engineering in 5 Domains: The Success Story of the MontiGem Generator
           Framework. Science of Computer Programming, 232:103033, 2024.

[Ch21]   Chen, Mark; Tworek, Jerry; Jun, Heewoo; Yuan, Qiming; Pinto, Henrique Ponde
           de Oliveira; Kaplan, Jared; Edwards, Harri; Burda, Yuri; Joseph, Nicholas; Brock-
           man, Greg et al.: Evaluating large language models trained on code. arXiv preprint
           arXiv:2107.03374, 2021.

[Ch23]   Class Diagram For Analysis. https://github.com/MontiCore/cd4analysis.

[Da22]   Dalibor, Manuela; Heithoff, Malte; Michael, Judith; Netz, Lukas; Pfeiffer, Jérôme;
           Rumpe, Bernhard; Varga, Simon; Wortmann, Andreas: Generating Customized Low-
           Code Development Platforms for Digital Twins. Journal of Computer Languages
           (COLA), 70, June 2022.

[De16]   Desai, Aditya; Gulwani, Sumit; Hingorani, Vineet; Jain, Nidhi; Karkare, Amey; Marron,
           Mark; R, Sailesh; Roy, Subhajit: Program synthesis using natural language. In: 38th Int.
           Conf. on Software Engineering. ICSE '16, ACM, NY, USA, pp. 345–356, May 2016.

[Er17]   Ernst, Michael D.: Natural Language is a Programming Language: Applying Natural
           Language Processing to Software Development. In: 2nd Summit on Advances in
           Programming Languages (SNAPL 2017). Schloss Dagstuhl - Leibniz-Zentrum fuer
           Informatik GmbH, 2017.

[FFK23]   Fill, Hans-Georg; Fettke, Peter; Köpke, Julius: Conceptual modeling and large language
           models: impressions from first experiments with ChatGPT. Enterprise Modelling and
           Information Systems Architectures (EMISAJ), 18:1–15, 2023.

[Fi22]   Finnie-Ansley, James; Denny, Paul; Becker, Brett A; Luxton-Reilly, Andrew; Prather,
           James: The robots are coming: Exploring the implications of openai codex on in-
           troductory programming. In: Australasian Computing Education Conf. pp. 10–19,
           2022.

[FL10]   Fowler, Martin; Languages, Domain-Specific: , Addison-Wesley Professional, 2010.

[FR07]   France, Robert; Rumpe, Bernhard: Model-driven Development of Complex Software:
           A Research Roadmap. Future of Software Engineering (FOSE '07), pp. 37–54, 2007.

[Ge20a]   Gerasimov, Arkadii; Heuser, Patricia; Ketteniß, Holger; Letmathe, Peter; Michael,
           Judith; Netz, Lukas; Rumpe, Bernhard; Varga, Simon: Generated Enterprise Information
           Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In: Comp.
           Proc. of Modellierung 2020 Short, Workshop and Tools & Demo Papers. CEUR
           Workshop Proceedings, pp. 22–30, 2020.

[Ge20b]   Gerasimov, Arkadii; Michael, Judith; Netz, Lukas; Rumpe, Bernhard; Varga, Simon:
           Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise
           Information Systems. In: 25th Am. Conf. on Information Systems (AMCIS 2020).
           AISeL. AIS, pp. 1–10, 2020.

[HKR21]   Hölldobler, Katrin; Kautz, Oliver; Rumpe, Bernhard: MontiCore Language Work-
            bench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software
            Engineering, Band 48. Shaker Verlag, May 2021.

[HMMM18]   Hernandez-Mendez, Adrian; Michel, Felix; Matthes, Florian: A practice-proven ref-
            erence architecture for model-based collaborative information systems. Enterprise
            Modelling and Information Systems Architectures (EMISAJ), 13:262–273, 2018.

[IA10]   Ibrahim, Mohd; Ahmad, Rodina: Class Diagram Extraction from Textual Requirements
            Using Natural Language Processing (NLP) Techniques. In: 2010 Second International
            Conference on Computer Research and Development. pp. 200–204, May 2010.

[Ja23]   Jalil, Sajed; Rafi, Suzzana; LaToza, Thomas D; Moran, Kevin; Lam, Wing: Chatgpt and
            software testing education: Promises & perils. arXiv preprint arXiv:2302.03287, 2023.

[JWM06]   John, Santhosh; Wi-Mei, Maisie Han: A proposed framework for an effective integra-
            tion of supporting environments for smart client application development. In: 2006
            International Conference on Computing & Informatics. IEEE, pp. 1–6, 2006.

[KRV08]   Krahn, Holger; Rumpe, Bernhard; Völkel, Steven: Monticore: Modular development of
            textual domain specific languages. In: International Conference on Objects, Components,
            Models and Patterns. Springer, pp. 297–315, 2008.

[KT08]   Kelly, Steven; Tolvanen, Juha-Pekka: Domain-specific modeling: enabling full code
            generation. John Wiley & Sons, 2008.

[Li22]   Li, Yujia; Choi, David; Chung, Junyoung; Kushman, Nate; Schrittwieser, Julian;
            Leblond, Rémi; Eccles, Tom; Keeling, James; Gimeno, Felix; Dal Lago, Agustin et al.:
            Competition-level code generation with alphacode. Science, 378(6624):1092–1097,
            2022.

[Mi19]   Michael, Judith; Netz, Lukas; Rumpe, Bernhard; Varga, Simon: Towards Privacy-
            Preserving IoT Systems Using Model Driven Engineering. In: MODELS 2019.
            Workshop MDE4IoT. CEUR-WS, pp. 595–614, 2019.

[Mi22]   Michael, Judith; Nachmann, Imke; Netz, Lukas; Rumpe, Bernhard; Stüber, Sebastian:
            Generating Digital Twin Cockpits for Parameter Management in the Engineering of
            Wind Turbines. In: Modellierung 2022. Gesellschaft für Informatik, pp. 33–48, 2022.

[MRZ21]   Michael, Judith; Rumpe, Bernhard; Zimmermann, Lukas Tim: Goal Modeling and
            MDSE for Behavior Assistance. In: Int. Conf. on Model Driven Engineering Languages
            and Systems Companion (MODELS-C). ACM/IEEE, pp. 370–379, 2021.

[Ob17]   OMG Unified Modeling Language (OMG UML). `https://www.omg.org/spec/UML/2.`
            `5.1/PDF`.

[Oh22]   Oh, Jaehoon; Kim, Sungnyun; Ho, Namgyu; Kim, Jin-Hwa; Song, Hwanjun; Yun, Se-
            Young: Understanding Cross-Domain Few-Shot Learning Based on Domain Similarity
            and Few-Shot Difficulty. In: Advances in Neural Information Processing Systems. 2022.

[Op23a]   OpenAI API. `https://platform.openai.com/docs/introduction`.

[Op23b]   OpenAI: , GPT-4 Technical Report, 2023.

[Pa20]   Pang, Xiongwen; Zhou, Yanqiang; Li, Pengcheng; Lin, Weiwei; Wu, Wentai; Wang,
            James Z.: A novel syntax-aware automatic graphics code generation with attention-based
            deep neural network. Journal of Network and Computer Applications, 161:102636,
            July 2020.

[RR16]   Richardson, Clay; Rymer, John R: The Forrester Wave™: low-code development
            platforms, Q2 2016. Forrester, Washington DC, 2016.

[Ru16]   Rumpe, Bernhard: Modeling with UML: Language, Concepts, Methods. Springer
            International, July 2016.

[Sa20a]   Sahay, Apurvanand; Indamutsa, Arsene; Di Ruscio, Davide; Pierantonio, Alfonso:
            Supporting the understanding and comparison of low-code development platforms. In:
            46th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA).
            IEEE, pp. 171–178, 2020.

[Sa20b]   Saini, Rijul: Artificial Intelligence Empowered Domain Modelling Bot. In: 23rd
            ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Compan-
            ion. MODELS '20, ACM, NY, USA, 2020.

[SVD21]   Shin, Richard; Van Durme, Benjamin: Few-shot semantic parsing with language models
            trained on code. arXiv preprint arXiv:2112.08696, 2021.

[Th22]   Thomas, Julien Joseph; Suresh, Vishnu; Anas, Muhammed; Sajeev, Sayu; Sunil, K. S.:
            Programming with Natural Languages: A Survey. In: Computer Networks and Inventive
            Communication Technologies. Lecture Notes on Data Engineering and Communications
            Technologies, Springer, Singapore, pp. 767–779, 2022.

[Ve20]   Veizaga, Alvaro; Alferez, Mauricio; Torre, Damiano; Sabetzadeh, Mehrdad; Briand,
            Lionel; Pitskhelauri, Elene: Leveraging Natural-Language Requirements for Deriving
            Better Acceptance Criteria from Models. In: ACM/IEEE Int.Conf. on Model Driven
            Engineering Languages and Systems. MODELS '20. ACM, p. 218–228, 2020.