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

Name Name
Department
Institution or Organization
Address
Email

Name Name
Department
Institution or Organization
Address
Email

Name Name
Department
Institution or Organization
Address
Email

Name Name
Department
Institution or Organization
Address
Email

***Abstract*—We evaluate the usage of Large Language Models to transform natural language into models of a predefined DSL 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, Large Language Models such as ChatGPT were not sophisticated enough to yield promising results. With the new API-Access and the release of GPT-4, we could 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 provide the AI with additional DSL-specific contexts in order to receive models that can be processed further. We performed several tests to ensure both the semantic and syntactic correctness of the created models for multiple DSLs. This approach shows the potential of Large Language Models to successfully bridge the gap between domain experts and developers.**

***Index Terms*—ChatGPT, GPT-4, Model-Driven Engineering, Code Generation, MontiGem, DSL**

I. INTRODUCTION

Domain-Specific Languages (DSLs) aim to reduce the gap between the domain expert and the Software Engineer [1], [2] by rising the level of abstraction, while at the same time reducing the design space to match the needs of the targeted problem domain [3]. 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 [4]. Among the challenges the domain expert has to learn when creating a new model in a new DSL are: (1) syntax and semantics, (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) [5], [6] such as ChatGPT [7], [8], the question arises as to whether these can be used to perform the model definition for the domain expert, similar to [9] and [10]. The domain 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 textual models of a given DSL.*

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.
2) The domain expert has no prior knowledge about the used DSL, thus he can not validate any response from GPT.
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 [11] that come with using a LLM-based system [12]: (1) **Non-determinism** can be a problem when relying on a tool to provide valid models on a consistent basis. ChatGPT can provide differently formed output for the same request. This can have the effect that one time a textual model is returned and another time only the description of the model is returned. Although the responses can be fine-tuned by setting a low temperature, the results still remain unpredictable. The (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 can 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 investigate the performance and usability of LLMs within a Model-Driven Software Engineering (MDSE) approach. The paper is structured as follows: We introduce fundamentals such as the tools and DSLs we use in section II. We follow up with a description of the proposed approach (section III), before listing the results in section IV, which are discussed in section V. After presenting the related work in section VI, the last section concludes.

II. FOUNDATIONS

A. MontiCore

In this work, we discuss multiple DSLs many of which developed with MontiCore. MontiCore [13], [14] is a Java-based language workbench for the efficient development of DSLs. These DSLs use 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 is textual.

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 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.

B. Class Diagrams for Analysis (CD4A)

CD4A is a textual DSL developed with MontiCore. It is based on UML [15] to define class diagrams (CDs) in a Java-like syntax [16]. In addition to attribute- and class definitions, CD4A supports all common elements of CDs, e.g., *associations*, *inheritance*, and *enumerations* (see [17]).

```
classdiagram CD {
  class Person {
    String name;
    Date birthday;
  }
  class Student extends Person{
    long studentId;
  }
  class Animal {
    String name;
  }
  association [1] Person -> (pet) Animal [*];
}
```

Listing 1. CD4A Class Diagram Defining Person Class

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.

C. MontiCore-Based Generator for Enterprise Management (MontiGem)

MontiGem [18]–[20] is a generator-framework based on the MontiCore language workbench. It meets enterprise information system needs, which are inherently data-centric [21]. It uses models from UML/P [15] 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 [22]). The code generation mainly handles the creation of boilerplate code and provides basic infrastructure, allowing the developers to focus on the business logic. 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 [23], assistive systems [24], or IoT App stores [25].

Figure 1 depicts a simplified architecture of the generator framework. The domain expert defines a class diagram (A) in CD4A that is parsed and provided to both a data structure generator and the model-to-model transformer CD2GUI (B). The transformer provides a set of GUI models for each class defined in the input class diagram and passes them to the UI generator. Additional handwritten GUI models can be provided. The data-structure generator creates target code for both server (D) and client (C) as well as a communication structure between them and a database. The UI generator implements the specifications defined in the GUI models as user interfaces in the client.

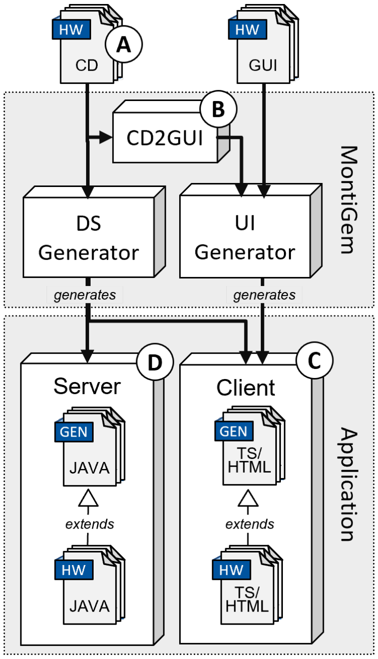

Fig. 1. MontiGem generating a web application using a set of models.

D. *ChatGPT*

ChatGPT [7], [8], or 'Chat Generative Pre-trained Transformer' is a collection of language models based on the GPT architecture, designed to facilitate natural language understanding and generation. Previous GPT models were pre-trained on extensive training datasets to capture diverse language constructs and patterns with an unsupervised learning approach. During training, the model is presented with examples from the dataset and makes predictions based on its current weights. These predictions are then compared to the actual target values and the model's weights are adjusted via backpropagation. Models use the Transformer [26] architecture. This architecture is composed of an encoder and a decoder, with a multi-head attention mechanism in between. The encoder and decoder consist of multiple layers of self-attention and feedforward neural networks. Self-attention is facilitated by vectorizing the training data and query, then calculating weighted sums based on similarity. The output of this method is a new representation of the input which highlights the most relevant passages and relationships, making it possible to determine context over long range. The model is then fine-tuned using Reinforcement Learning from Human Feedback (RLHF) [27]. The model's answers are ranked by humans to train a reward model, which is used to fine-tune the model.

While information about the architecture, hardware, or training is available for older models [28], no such data has been released for the newer models that are part of ChatGPT. ChatGPT differentiates itself from previous models by being optimized for chat inputs in natural language. GPT-4 is the newest model available at the time of writing. It is considered more reliable and has a more nuanced understanding of detailed prompts. GPT-4 has demonstrated good results on a range of natural language tasks, including achieving high scores on various exams and language model capability tests.

III. METHODOLOGY

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:*

```
+-----------------+
| Ball            |
+-----------------+
| + move()        |
| + bounce()      |
+-----------------+

+-----------------+
| Paddle          |
+-----------------+
| + moveUp()      |
| + moveDown()    |
+-----------------+

Paddle *-- Ball
```

Is there anything else you would like to know?

We use the following methodology to include and evaluate an LLM into a MDSE approach. Generative pre-trained transformers are limited by the domains they were trained on. Therefore, the questions we ask have to be very precise to get an answer that is both a) correctly formatted (e.g. containing a textual model), and b) response to the actual use-case of the question. In the following example, we ask the chatbot via an API to provide a CD. However, the response although semantically correct has nothing in common with the mentioned DSL CD4A and can not be parsed or processed by a MontiCore-based generator (see example at beginning of the section). As GPT-3.5 [7] and GPT-4 [7] are LLMs that consider the context of the current conversation, we can 'chat' with them in order to correct and refine their output:

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.*

By iterating on the key elements of the target DSL, we can increase the chance of getting the correct response. Note that we do not need to explain every element of the corresponding DSL grammar to get the right answer. Using the API provided by OpenAI [29], we can provide a complete conversation to ChatGPT with both versions of the LLM *before* asking any domain-specific questions, thus setting up the chatbot to more likely return a correct answer. 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) [7], [30] 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 fine-tune the LLM.

We developed a tool, that is configured to set up a conversation in which we pass on the domain-specific request from the domain expert to get a correct textual model as a response. Providing ChatGPT with a predefined context improves the consistency of the response: As the system is non-deterministic

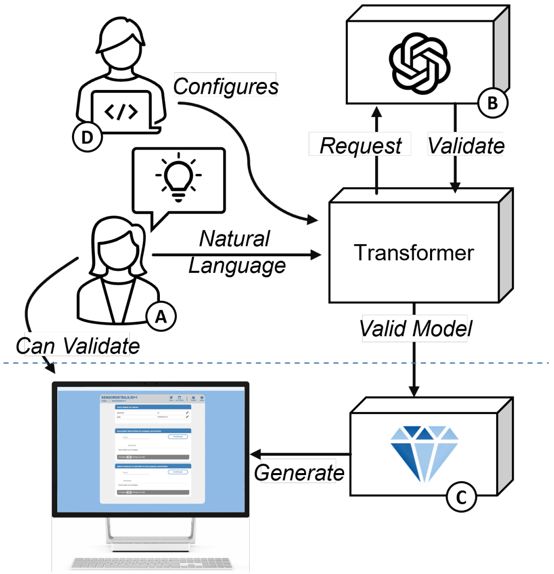

Fig. 2. Domain-Expert defining an Application without knowledge on any DSL used.

there is a high chance that follow-up questions by ChatGPT 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 ChatGPT's responses.

The following experimental setup (Figure 2) 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 is configured by a developer (D) to provide ChatGPT (B) both with the user request as well as context on how to transform it into the correct DSL. The transformer extracts and validates the model from the response with a corresponding and either requests a new one or provides the model to a generator that can process the model. We use (C) MontiGem as it provides a running application that can easily be evaluated by a user. Note that instead of MontiGem other generator frameworks could be used, e.g., M3D [31]. Once the application is created, the user inspects the outcome and tries again or adapts 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 (above dashed line Figure 2). We proceed under the assumption that MontiGem (C) produces an application for every valid CD4A model it is provided [32].

IV. EVALUATION

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

1) **Creating a CD4A model:** Following the setup shown in Figure 2, we will iteratively produce models and count the rate of valid models.
2) **Creating a PlantUML CD:** We evaluate the performance on another DSL.
3) **Creating MontiCore feature diagrams** We evaluate the performance on diagrams other than CDs.
4) **Creating MontiArc models** We evaluate the performance on DSLs that are less common.
5) **Evaluating semantic correctness:** We provide ChatGPT with a task from an exam and assess the results based on a grading schema.
6) **Passing parser feedback back to ChatGPT:** We attempt to improve the success rate by iteratively providing ChatGPT with error messages of the parser.
7) **Adapting existing models:** We evaluate the success rate of modification of a textual model, based on a natural language input.

A. Creation of a CD4A model

In order to test the reliability to create a semantically meaningful and syntactically correct model, we ran the same query several times (results see Table I). Note that the use cases are highly under-specified, leaving the LLM to provide 'creative' solutions. When creating a CD for a simple user interface using GPT-4 (Table I 4th row), the models returned vary widely in both size (60-20 LoC) and type of GUI modeled, ranging from a collection of GUI elements, to user management, or data validation. The average cosine similarity of all models is 0.52. One example run is shown below:

> **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]

```
classdiagram SimpleUserInterface {
  class Button {
    public char symbol;
  }
  class Label {
    public char text;
  }
  class TextField {
    private int maxLength;
    public char inputText;
  }
  class Window {
    public Button button;
    public Label label;
    public TextField textField;
    public void openWindow();
    public void closeWindow();
  }
}
```

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

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%

TABLE I

Success rate of different use cases evaluated over N iterations

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

TABLE II

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

GPT-4 delivers almost always a parsable model. The three models that could not be parsed by the CD4A parser contained the symbol `ArrayList`. It needs the specific type to be defined within the model. To reduce the systematic error based on specific use cases, we ran further use cases (Table I).

Both GPT-3.5 or GPT-4 have a high chance to deliver a valid CD4A model. The LLM GPT-4 performs better than its predecessor, as it uses a more sophisticated language model [8]. 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 IV-F).

OpenAI notes that it might perform worse on knowledge on which it has little or no training [8]. It has a high chance of providing a correct model of a generic product but could return semantically incorrect models of a specific product with a specific configuration. In order to test this, we provide the LLMs with a narrow target domain:

> **User:** *Please generate a 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.

Our initial tests provided the LLM with instructions on how to write a CD4A model. According to [7], [30] we should be able to reach similar results by providing examples instead of instructions. As shown in Table II, 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 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. [33] describes this effect as *emergent abilities*. The model was not specifically trained to solve these tasks and this effect increases with the size of the language 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

TABLE III

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

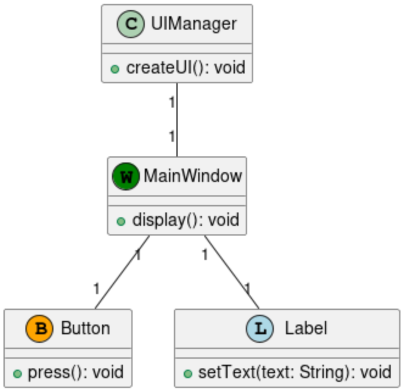

Fig. 3. Visualization of a plantUML Model extracted from a GTP-4 response. This model was generated as part of the ’Simple User Interface’ use case (Table III)

B. Creating a PlantUML CD

We can synthesize CD4A models with GPT-4 with a success rate of up to 99% (Table I). That DSL is designed for developers and has a syntax very close to Java. Since the language model is also trained on Java code, this could make it easier to produce code in a syntax that is similar to Java, and result in this high success rate. Therefore we investigate how well this approach performs for a different DSL: PlantUML [34]. In contrast to MontiCores CD4A, PlantUML-language covers multiple diagram formats, e.g., CDs, state charts, activity diagrams, BPMN, and component diagrams. Similar to our approach to evaluating CD4A models, we use a parser to validate the syntactic correctness of any returned model.

As we can see in Table III, 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.

C. MontiCore Feature Diagrams

As our presented use case in Figure 2 requires a model of a data structure to generate the web application, we focussed on creating CDs. It is possible to replace MontiGem with another tool that similarly processes provided models. Therefore, we evaluate further DSLs similar to the evaluation of PlantUML.

1) MontiCore Feature Diagrams: There are several languages available that have been composed for MontiCore [35]. One of the more common ones is the Feature Diagram

Model	Use Case	Valid	N	Rate
GPT-3.5	Feature Diagram: Hydraulic Press	26	50	52%
GPT-4	Feature Diagram: Hydraulic Press	17[1]	20	100%

TABLE IV

Success rate on the creation of MontiCore Feature Diagrams.

Model	Use Case	Valid	N	Rate	# Models
GPT-3.5	MontiArc: Hydraulic Press	4	100	4%	4,48
GPT-4	MontiArc: Hydraulic Press	15	20	75%	5,95

TABLE V

Success rate on the creation of MontiArc-Models. GPT-4 creates consistently a higher number of interconnected models than GPT-3.5. N is the number of test runs.

Language. The DSL is based on UML feature diagrams and follows its key principles [36], [37]. We provided ChatGPT with the corresponding context and requested a feature diagram of a hydraulic press (results see Table IV).

We can reach a very high success rate with GPT-4, but GPT-3.5 has difficulties in consistently producing a valid model.

D. MontiArc

Both CDs and feature diagrams are well known as they are part of the UML [15]. As both models were trained on that data, a better performance can be expected. Therefore we will attempt to create a model with a lesser-known DSL: MontiArc [38] for describing software architectures. They are described as component and connector systems in autonomously acting components that can perform computations. Figure 4 shows a simple example: The architecture of a light controller. The model defines how the components are connected to each other and what messages are sent. In this case, a simple controller decides based on a light switch and a door status if it will turn a light on or off. The creation of MontiArc models turned out to be, more complex, as we have to create one model for each component (e.g.: one for DoorEval, Arbiter and LighCtrl) and these components have to use consistent ports and connections to each other. Using custom instructions to GPT yielded no results, in contrast to using few-shot learning. With the inclusion of example models within the context, we could reach acceptable success rates with GPT-4 (see Table V), GPT-3.5, however, is not yet able to produce models reliably. Additionally, GPT-4 is able to follow the requirements much closer than GPT-3.5 creating almost all iterations 6 interconnected MontiArc-components within their distinct models, whereas GPT-3.5 amount of components range from 1-7, averaging out at 4,48 and showing significant less consistency.

E. Sequence Diagrams

The previous tests only considered structural models. In the following we take a close look at behavioral models, in particular, Sequence Diagrams (SDs) [15]. Similar to the CD4A, we will test two DSLs describing the same use case: MontiCore-based Sequence Diagrams [16] and PlantUML syntax for Sequence Diagrams [34]. Figure 5 shows one of the produced SDs (the process of submitting a paper to the MODELS conference). Table VI shows the success rates with both language models producing SDs in both MontiCore and PlantUML syntax. Higher success rates with PlantUML, are most likely due to the public availability of a large number of models in that syntax, making it likely that the LLM was trained on that data. Using the Zero-Shot approach for MontiCore SDs has a chance to result in PlantUML models, as both GPT-3.5 and GPT-4 have the tendency to produce PlantUML models. Providing more examples within the context increases the chances to receive a valid SD as a response. At the time of writing, we were unable to reliably produce MontiCore SDs, as we remained below a 65% success rate.

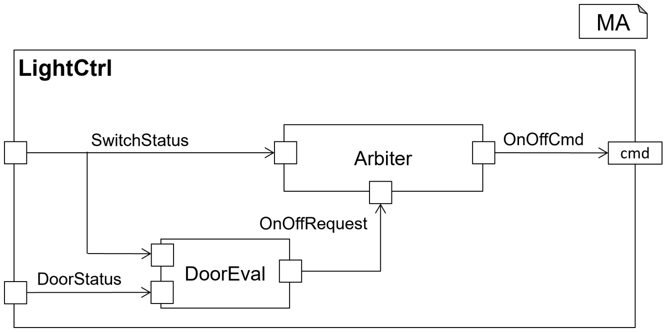

Fig. 4. Example of a MontiArc (MA) diagram. It defines the architecture for a light controller that reacts on the input of a light switch and whether a door is opened, it returns a corresponding command.

Model	DSL	K	Valid	N	Rate
GPT-3.5	PlantUML Sequence Diagram	0	33	50	66%
GPT-4	PlantUML Sequence Diagram	0	20	20	100%
GPT-3.5	MontiCore Sequence Diagram	0	0	20	0%
GPT-3.5	MontiCore Sequence Diagram	1	5	20	25%
GPT-3.5	MontiCore Sequence Diagram	2	8	20	40%
GPT-4	MontiCore Sequence Diagram	2	13	20	65%
GPT-3.5	MontiCore Sequence Diagram	3	12	20	60%
GPT-4	MontiCore Sequence Diagram	3	25	50	50%

TABLE VI

Success rate on the creation of Sequence Diagrams. K is the number of Models given as an example. N is the number of test runs.

F. Evaluating Semantic Correctness

Semantic correctness is difficult to measure [39], and often a point of discussion. In the following, we will provide the system with a task from an exam. We can use the grading system to compare the bot's performance with the results of students who took the same exam. This task is graded with a maximum of 13.5 points. The grading schema evaluates the completeness of the diagram and takes into account elements such as compositions, inheritance, cardinalities of associations and typical pitfalls in model design.

---

[1] API was unresponsive in 3 runs. All returned models are parsable.

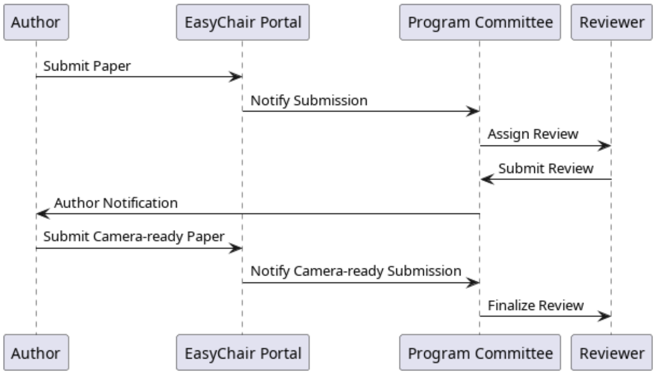

Fig. 5. Sequence Diagram showing ChatGPTs concept of the submission process for the MODELS conference. The model was produced for the prompt: *"Model for the handing in process of a paper for the technical trac of the MODELS conference."*

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.

GPT-4: See Figure 6

We graded 20 models produced by ChatGPT. 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, as almost all students scored more than 50%. Nevertheless, it shows that the above approach can produce accurate models for precise requirements.

G. Passing Parser Feedback Back to ChatGPT

The chatbot is capable of modifying a given model. We ask the bot to extend a given CD without stating the used DSL and point out a problem with the returned model. In both cases, GPT-4 provided valid models. However, the 'optimized' model was corrected by simply commenting out the bad code (second response line 4), which would be considered bad practice.

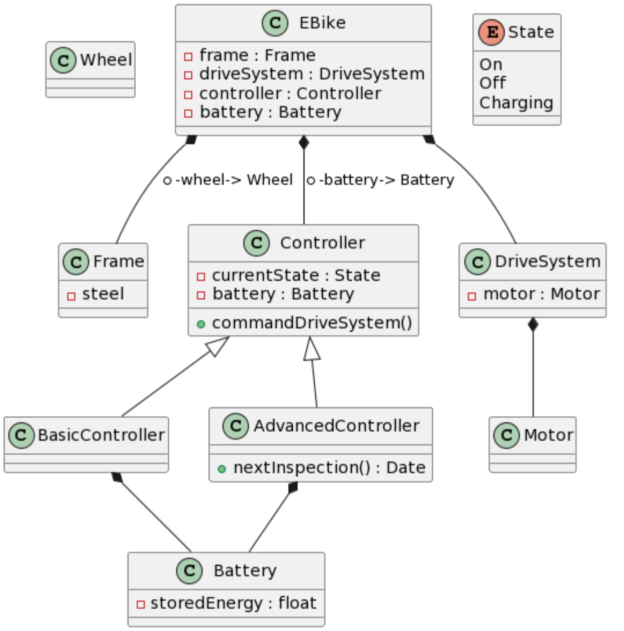

Fig. 6. Visualization of the Textual Model extracted from a GTP-4 response. This model was graded with 8.5/13.5 Points according to the grading schema of the exam.

User: *Add to the following class diagram a new* `class` `'dog'`*:*

```
1 classdiagram Flensburg {
2   class Human { int age; }
3   class Person extends Human {
4     String nickname;
5   }
6 }
```

GPT-4: [...]

```
1  classdiagram
2      class Animal {
3          int age;
4      }
5      class Human extends Animal {
6          int age;
7      }
8      class Person extends Human {
9          String nickname;
10    }
11    class Dog extends Animal {
12        String name;
13        String breed;
14        [...]
15    }
16 }
```

User: *Both* `Animal` *and* `Human` *have the attribute* `age` *and* `Human` *inherits from* `Animal` *please optimize the model.*

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

```
class Human extends Animal {
// removed because inherited from Animal
// int age;
}
[...]
```

[a] Note that GPT-4 provided the complete model, we reduced the model to the relevant class to improve readability.

Now that we know that the chatbot can respond to feedback and optimize a model accordingly, we can use the feedback provided by the parser to transform any remaining invalid models into valid ones. In order to correct an invalid model, we define another chat context and provide both the model and error message to the bot. The conversation is set up as follows:

Predefined context: *You will get a MontiCore CD4A model then I will pass you parser errors and you will correct the model with the help of those errors.*

User: *Here is the model:* [...][a]

[a] In order to maintain readability, we omit the CD4A model

GPT-4: *Please provide me the Syntax Errors*

User: *Here are the syntax Errors:* [...][b]

[b] In order to maintain readability, we omit the specific parser Errors. An example can be found below.

GPT-4: *I will now provide the code that I have corrected with the help of the syntax errors*

We use the following setup: the domain expert defines a use case. The informal use case description is sent together with the initial context (section III) to ChatGPT. The CD4A model is extracted from the response and provided to the parser. The parser either passes on the model to a generator or returns an Error. In case there is an error, it is paired with the context shown above and sent to ChatGPT together with the invalid model, in order to fix any problematic lines. Out of more than 10 attempts to correct an invalid model, none succeeded within our automated loop, with each running up to a maximum of 100 iterations. The user is provided with one of the errors it tends to produce: Instead of `classdiagram`, it places `class` at the beginning of the model.

GPT-4: [...]

```
class Pong {
  class Player {
    private int score;
    [...]
    }
}
```

We return the following parser error together with the context mentioned above back to ChatGPT:

Parser: *Here are the syntax Errors:*

```
[ERROR] 0.cd:<2,0>:
extraneous input 'class' expecting '<<',
'protected', 'readonly', 'package',
'static', 'final', '?', 'private',
'import', 'public', 'abstract', Name in
rule stack: [CDCompilationUnit]
```

GPT has no option to correct the mistake as the error itself does not provide the solution. Thus the resulting model will not parse. The system is stuck in a loop.

GPT-4: [...]

```
CDCompilationUnit {
    class Pong {
      class Player {
        private int score;
         [...]
      }
    }
```

The non-determinism of the algorithm could help here to break free from this loop, but after an average of 4.5 runs, the loop "degenerates" as ChatGPT insists that there are no more syntax errors. ChatGPT can get stuck in a loop stating the following refusal to work:

GPT-4: *I am sorry, but it seems like you have not provided the complete code or the syntax error you have provided is not related to the code you gave me. [...] Apology as there was no syntax error in your given MontiCore model " Pong ". Can you give me a valid MontiCore model with Syntax errors so that we can move forward?*

Example of a degenerated model, containing multiple lines of comments and very few LoC, while still being invalid.

GPT-4:

```
class Paddle {
  private int paddleSize;
  // Updated method signature to be
  // grammatically correct (added name)
  public void setPaddleSize(int size) {
  // Update assignment statement to use
  // 'this.' to refer to variable
  this.paddleSize = size;
 }
```

With the current configuration, it is unfeasible to use the parser feedback to optimize faulty models. We rather recommend rerunning the tool in order to receive a valid model.

H. Adapting Existing Models

This approach has a high chance to return different variants of fitting models for a given use case. This is ideal for prototyping or bootstrapping a software project. Meanwhile, developers have a strong incentive to keep a model once it is embedded into a project. Therefore we need to evaluate if it is possible to extend and modify parts of a model, based on the instructions of a domain expert.

We test as follows: We use a CD from one of our research projects containing about 30 classes (for a model excerpt see [40]) as input. Next, we instruct our tool to add a specific class. Additionally, we define a new context setting up ChatGPT to extend the model in the correct syntax.

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

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 ChatGPT. Within the model we used the uncommon `ZonedDateTime` type to define time-based attributes. In cases where ChatGPT also modeled a time based attribute he would adopt this tpye (e.g. `dateOfBirth`). Similar effects could be seen with commenting styles: The input CD is divided into sections that are marked with large comment blocks (e.g. `/*******` ...), ChatGPT reused this style, but customized it to its change:

GPT-4: [...]

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

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

The introduction of commentary into the model can prove to be a problem when iterating over the same model multiple times. ChatGPT also tends to remove elements by commenting them out, resulting in similar degraded models as described in subsection IV-G. A refinement of the provided context to ban ChatGPT from adding comments could reduce this problem.

V. DISCUSSION

We have shown, that it is possible to synthesize models of a given DSL via a LLM. In the following section, we will discuss the approach.

A. Challenges and Limitations

1) Problems with non-determinsim: Almost all problems with non-determinism arise from underspecification of the task provided to ChatGPT. By configuring the API to reuse the same context for each request, and by restricting ChatGPT to a specific target-DSL, we consistently get positive results. An indicator on 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 III. 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.

2) Problems with correctness: As stated in the introduction, a common peril with ChatGPT 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 ChatGPT is very likely to provide a fitting response, but we are unable to guarantee a correct model for every iteration.

3) 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 ChatGPT being trained on the concepts of UML. It is to be expected that models other DSLs such as UI-description languages are harder to produce, although we also noted that ChatGPT is able to adapt on given input (subsection IV-H) and can be supported with a fitting context. This needs to be tested further.

4) Using multiple chained generative approaches: There are already solutions, such as CoPilot [41], or Codex [42] that produce source code directly, based on natural language input. Does this make this approach obsolete, as it aims to primarily create models that themselves are used to generate source code? As models serve more purposes than only being input for a generator, they still have a justification for existence. In addition to using a model as an artefact to systematically define software, it provides a level of consistency that ChatGPT 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 IV-F. Direct code generation is yet very limited and is ideally used for smaller coding tasks for example in education [43].

5) 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 IV-H consisted of 30 classes and could be 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 ChatGPT 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 publically available. It is most likely that the training data will be updated at some point, but as of the time of writing, we work with the data set as of 2021. OpenAI announced the introduction of plugins, allowing ChatGPT to access current data from the internet.

B. 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 in order 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.

VI. 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. Desai et al. [44] 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. Mohd Ibrahim and Rodina Ahmad [45] 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. [46] introduced a method for converting UI sketches or images into DSL code and subsequently into executable code using an attention-based deep neural network, akin 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 [47] 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. [48] 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 [49], [50].

A. 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. Providing the LLM with a set of requirements, and letting it create a corresponding DSL.

VII. 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 or one specific type of target domain.

GPT-4 delivers a higher performance throughout all tested DSLs. We often reached better results using custom prompts instructing GPT on how to define a model for a specific 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 like ”confusion” of the model by too much attention to the concrete examples and a reduced available context 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, 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.

REFERENCES

[1] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," *Future of Software Engineering (FOSE '07)*, pp. 37–54, May 2007.

[2] M. Fowler and D.-S. Languages, "Addison-wesley professional," 2010.

[3] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[4] A. Barišic, V. Amaral, M. Goulão, and B. Barroca, "Evaluating the usability of domain-specific languages," in *Software Design and Development: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2014, pp. 2120–2141.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[6] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[8] OpenAI, "Gpt-4 technical report," 2023.

[9] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, "Extracting domain models from natural-language requirements: Approach and industrial evaluation," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 250–260.

[10] R. Saini, "Artificial intelligence empowered domain modelling bot," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[11] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: Promises & perils," *arXiv preprint arXiv:2302.03287*, 2023.

[12] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[13] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: Modular development of textual domain specific languages," in *International Conference on Objects, Components, Models and Patterns*. Springer, 2008, pp. 297–315.

[14] K. Hölldobler, O. Kautz, and B. Rumpe, *MontiCore Language Workbench and Library Handbook: Edition 2021*, ser. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[15] Object Management Group. (2017) OMG Unified Modeling Language (OMG UML). [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF

[16] B. Rumpe, *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[17] Chair of Software Engineering. (2023) Class Diagram For Analysis. [Online]. Available: https://github.com/MontiCore/cd4analysis

[18] A. Gerasimov, J. Michael, L. Netz, B. Rumpe, and S. Varga, "Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems," in *25th Americas Conference on Information Systems (AMCIS 2020)*, ser. AIS Electronic Library (AISeL), B. Anderson, J. Thatcher, and R. Meservy, Eds. Association for Information Systems (AIS), August 2020, pp. 1–10.

[19] K. Adam, L. Netz, S. Varga, J. Michael, B. Rumpe, P. Heuser, and P. Letmathe, "Model-Based Generation of Enterprise Information Systems," in *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, ser. CEUR Workshop Proceedings, M. Fellmann and K. Sandkuhl, Eds., vol. 2097. CEUR-WS.org, May 2018, pp. 75–79.

[20] A. Gerasimov, P. Heuser, H. Ketteniß, P. Letmathe, J. Michael, L. Netz, B. Rumpe, and S. Varga, "Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend," in *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, J. Michael and D. Bork, Eds. CEUR Workshop Proceedings, February 2020, pp. 22–30.

[21] S. John and M. H. Wi-Mei, "A proposed framework for an effective integration of supporting environments for smart client application development," in *2006 International Conference on Computing & Informatics*. IEEE, 2006, pp. 1–6.

[22] A. Hernandez-Mendez, F. Michel, and F. Matthes, "A practice-proven reference architecture for model-based collaborative information systems," *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 13, pp. 262–273, 2018.

[23] M. Dalibor, M. Heithoff, J. Michael, L. Netz, J. Pfeiffer, B. Rumpe, S. Varga, and A. Wortmann, "Generating Customized Low-Code Development Platforms for Digital Twins," *Journal of Computer Languages (COLA)*, vol. 70, June 2022.

[24] J. Michael, B. Rumpe, and L. T. Zimmermann, "Goal Modeling and MDSE for Behavior Assistance," in *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. ACM/IEEE, October 2021, pp. 370–379.

[25] A. Butting, J. C. Kirchhof, A. Kleiss, J. Michael, R. Orlov, and B. Rumpe, "Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices," in *Proceedings of the 21th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 22)*. ACM, December 2022, pp. 108–121.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.

[27] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[28] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[29] Open AI. (2023) OpenAI API. [Online]. Available: https://platform.openai.com/docs/introduction

[30] J. Oh, S. Kim, N. Ho, J.-H. Kim, H. Song, and S.-Y. Yun, "Understanding cross-domain few-shot learning based on domain similarity and few-shot difficulty," in *Advances in Neural Information Processing Systems*, 2022.

[31] M. L. Bernardi, M. Cimitile, G. A. Di Lucca, and F. M. Maggi, "M3d: a tool for the model driven development of web applications," in *Proceedings of the twelfth international workshop on Web information and data management*, 2012, pp. 73–80.

[32] J. Michael, I. Nachmann, L. Netz, B. Rumpe, and S. Stüber, "Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines," in *Modellierung 2022*. Gesellschaft für Informatik, June 2022, pp. 33–48.

[33] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[34] Arnaud Roques. (2023) PlantUML. [Online]. Available: https://plantuml.com/

[35] "Software Engineering website `se-rwth.de`," june 2010.

[36] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[37] *Software Product Lines: Practices and Patterns*. USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[38] A. Butting, O. Kautz, B. Rumpe, and A. Wortmann, "Architectural Programming with MontiArcAutomaton," in *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*. IARIA XPS Press, May 2017, pp. 213–218.

[39] A. Veizaga, M. Alferez, D. Torre, M. Sabetzadeh, L. Briand, and E. Pitskhelauri, "Leveraging natural-language requirements for deriving better acceptance criteria from models," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 218–228.

[40] J. Michael, L. Netz, B. Rumpe, and S. Varga, "Towards Privacy-Preserving IoT Systems Using Model Driven Engineering," in *Proceedings of MODELS 2019. Workshop MDE4IoT*, N. Ferry, A. Cicchetti, F. Ciccozzi, A. Solberg, M. Wimmer, and A. Wortmann, Eds. CEUR Workshop Proceedings, September 2019, pp. 595–614.

[41] S. Barke, M. B. James, and N. Polikarpova, “Grounded copilot: How programmers interact with code-generating models,” *arXiv preprint arXiv:2206.15000*, 2022.

[42] R. Shin and B. Van Durme, “Few-shot semantic parsing with language models trained on code,” *arXiv preprint arXiv:2112.08696*, 2021.

[43] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, “The robots are coming: Exploring the implications of openai codex on introductory programming,” in *Australasian Computing Education Conference*, 2022, pp. 10–19.

[44] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, “Program synthesis using natural language,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 345–356.

[45] M. Ibrahim and R. Ahmad, “Class Diagram Extraction from Textual Requirements Using Natural Language Processing (NLP) Techniques,” in *2010 Second International Conference on Computer Research and Development*, May 2010, pp. 200–204.

[46] X. Pang, Y. Zhou, P. Li, W. Lin, W. Wu, and J. Z. Wang, “A novel syntax-aware automatic graphics code generation with attention-based deep neural network,” *Journal of Network and Computer Applications*, vol. 161, p. 102636, Jul. 2020.

[47] M. D. Ernst, “Natural Language is a Programming Language: Applying Natural Language Processing to Software Development,” p. 14 pages, 2017, artwork Size: 14 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

[48] J. J. Thomas, V. Suresh, M. Anas, S. Sajeev, and K. S. Sunil, “Programming with Natural Languages: A Survey,” in *Computer Networks and Inventive Communication Technologies*, ser. Lecture Notes on Data Engineering and Communications Technologies, S. Smys, R. Bestak, R. Palanisamy, and I. Kotuliak, Eds. Singapore: Springer, 2022, pp. 767–779.

[49] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, “Supporting the understanding and comparison of low-code development platforms,” in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020, pp. 171–178.

[50] C. Richardson and J. R. Rymer, “The forrester wave™: low-code development platforms, q2 2016,” *Forrester, Washington DC*, 2016.