

Conceptual Modeling and Large Language Models: Impressions From First Experiments With ChatGPT

Hans-Georg Fill^{*,a}, Peter Fettke^b, Julius Köpke^c

^a University of Fribourg, Switzerland

^b Saarland University and German Research Center for Artificial Intelligence (DFKI), Germany

^c University of Klagenfurt, Austria

1 Motivation

Since OpenAI publicly released ChatGPT in November 2022¹, many ideas have emerged as to which applications this type of technology could support. At its core, ChatGPT is a *conversational artificial intelligence*, meaning that it can engage in a dialogue to respond to user input given in natural language (Campbell 2020). Although such types of systems have been well-known since Weizenbaum's Eliza program (Weizenbaum 1966) and are today widely deployed in practice under the popular term *chatbots*, ChatGPT has a particular set of properties that contributed to its wide reception and the recent hype surrounding it.

In contrast to previous chatbots, ChatGPT does not retrieve responses from a knowledge base, which has been pre-defined by some human user. Rather, it is based on a pre-trained generative language model, which creates responses based on patterns that the user supplies as input. Thereby, a language model basically assigns probabilities to every word in a vocabulary that can follow a given input sequence. Such word embeddings are trained using artificial neural networks to learn a probability distribution from given texts in an unsupervised fashion, i. e. such that no additional human input or labeling is required. The generation of the output sequence thereby considers the tokens of the input sequence and their position as well as the previously generated output,

which is thus denoted as an *autoregressive generation* (Jurafsky and Martin 2023). For the training of these probability distributions for the positional word embeddings, large sets of training data are required. In the case of the GPT-3 model, which underlies ChatGPT, this amounted to 175 billion parameters (Brown et al. 2020). For efficiently handling such large parameter sets, several innovations such as the architecture of transformer models (Vaswani et al. 2017) were necessary.

What seems to make ChatGPT however outperform large language model (LLM) based programs that had been released to the public previously, is its ability to reduce toxic outputs, i. e. harmful or biased results. This has been achieved through the approach of InstructGPT (Ouyang et al. 2022), which uses reinforcement learning from human feedback to train a reward model. This reward model is then in turn used to fine-tune the output generated by the GPT-3 and GPT-4 language models. Thus, the training of the reward model only requires rather limited resources compared to the size of the language model.

From the multitude of areas in which the application of ChatGPT is currently discussed, the following shall serve as non-exhaustive examples, which have already appeared in academic outlets: In the medical domain, ChatGPT has been considered for writing patient clinic letters by giving it instructions for following specific directions, using national guidelines and data from these guidelines in order to derive clinical advice (Ali et al. 2023); in the legal domain, ChatGPT has been asked to

* Corresponding author.

E-mail. hans-georg.fill@unifr.ch

¹ <https://openai.com/blog/chatgpt>

produce answers to final exams for law school courses (Choi et al. 2023) and in information management, a large number of scholars has compiled thoughts on different aspects of the impact and societal effects of ChatGPT (Dwivedi et al. 2023). ChatGPT has been proposed for improving code quality, refactoring, requirements elicitation, and software design in (White et al. 2023). In software development, approaches based on earlier variants of generative AI models are already available as commercial products, such as GitHub's copilot, (GitHub 2021) and gained interest in the research community (Nguyen and Nadi 2022; Pearce et al. 2022).

In the field of conceptual modeling, we have not yet come across any sources of using ChatGPT. Previous approaches only studied traditional knowledge-based chatbots, e. g. (Ren et al. 2020). The only related approach known so far is the one of GraphGPT, where simple types of graphs are derived via the ChatGPT API and visualized².

For the interaction with ChatGPT, there is currently no kind of instruction set or user manual available. Rather, the knowledge how to interact with it and which practices work best for achieving particular results are under exploration.

Thus, we will describe in the following several experiments we have conducted using ChatGPT based on the most recent GPT-4 models to explore which potential applications can be imagined in the future for generating and interpreting conceptual models. Thereby, we explore how to generate and interpret ER, Business Process, and UML class diagrams as well as HERAKLIT models. Our goal is to highlight how modeling applications based on large language models such as ChatGPT could be realized. We do not claim a full coverage of all functionalities, but rather aim to inspire the community to build on our examples and propose their own approaches.

² <https://github.com/varunshenoy/GraphGPT> last accessed 2023-03-16

2 Results from First Experiments

In the following sections, we report on the results of a series of experiments that we conducted to investigate how concrete prompts may be engineered for generating and interpreting conceptual models. The prompts are based on templates inspired by GraphGPT that first set the context and then include a variable for a concrete task definition. We will only show the task definition together with the results. The architecture, detailed structure of the prompts and the code can be found in the Appendix.

2.1 Experiment 1: Entity-Relationship Diagrams

The first modeling language we considered were Entity Relationship diagrams. These are not only one of the most well-known types of conceptual models but are comparatively simple in terms of the number of contained elements and relations, as well as their syntax. Thus, we started by designing an according prompt template.

The details of this template are included in Prompt 3 in the Appendix on Page 12. The prompt starts with a brief explanation of ER diagrams. Throughout the experiments, we noted that it is beneficial to explicitly specify the range of possible cardinalities. Subsequently, an example for an ER diagram based on this definition is presented in JSON syntax. Finally, a concrete output instruction is added to the prompt template, whereby the formatting is specified so that ChatGPT only generates the plain JSON output, which can be used for further processing. The last part in the prompt template is a variable denoted by `$code`. There, the actual task definition is inserted. An example for such a definition is shown in Task Definition 1.

Task Definition 1

Hotels have at least one or several rooms and each room is in turn part of exactly one hotel. Each room has a room number. Guests may stay in a hotel and book one or several rooms. A room may in turn be booked by none or more guests.

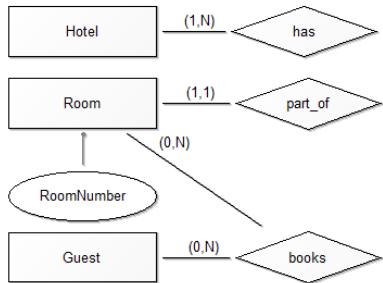


Figure 1: Graphical Representation of the ER Diagram Result Generated by ChatGPT Using GPT-4

The diagram in Fig. 1 is a graphical representation of the result generated by ChatGPT in a simplified JSON format, as shown in Prompt Result 2 in the Appendix on page 12. The solution generated by ChatGPT corresponds to the format proposed in the prompt template, which would make it suitable for further automated processing. The content of the result corresponds largely to a correct solution, i. e. the derived entity types are correct, the attribute and the attribute assignment. A slight error has been introduced for the relationship types where it created two separate relationship types *has* and *part_of*, which should be combined to one single type. The cardinalities are however correct.

In the next step, we moved on to the interpretation of existing diagrams. For this purpose, we reused the main part of the prompt template from before and modified the instruction part at the end, thereby asking ChatGPT to interpret a provided ER diagram in the explained notation in a textual format. The details are included in the template in Prompt 4 on page 12 in the Appendix.

We then added a problem definition in the form of a simplified diagram description in our custom notation, as shown here:

This led to the output shown in the following Prompt Result 1. The interesting aspect thereby is that ChatGPT detected an inconsistency in our problem definition. This was not done on purpose, but was actually an error we had coincidentally introduced in the translation to JSON format. We decided to keep it for illustration as it nicely shows the capabilities of ChatGPT, i. e. to note that an

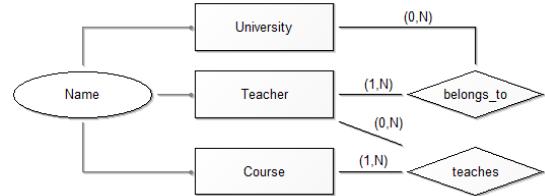


Figure 2: ER Diagram Input for Model Interpretation

attribute has been defined for an entity type that has not been specified previously.

Prompt Result 1

This ER diagram describes a university system with three main entity types: University, Teacher, and Course. The relationship types in this system are "teaches" and "belongs_to". A teacher can belong to one or many universities, while a university can have zero or many teachers. A teacher can teach zero or many courses, while a course must be taught by at least one teacher.

Attributes are also defined for each entity type: Name for University and Teacher, and Name for Course (although it is not clear if "Student" is also an entity type, as it is not included in the "EntityTypes" list, but it has an attribute named "Name" in the "has_attribute" section).

In summary, this ER diagram represents a university system where universities have teachers who teach courses, and each of these entities has a name attribute.

The textual description further correctly describes the entity types, the cardinalities and the relationship types. It also adds a description of the overall scenario at the end of the output. In our view, the interpretation of the diagram seems to be better than the generation, as no obvious errors have been introduced. On the other hand, the exact definition of the semantics of ER diagrams would certainly require additional information to the one we integrated in the prompt. For example, it has not been explained so far how the derivation of relationship types actually works. Still, the results can serve as a good basis for subsequent refinement steps. Such refinements could be done, for example, to instruct ChatGPT to add relationship types, remove connections or add attributes. In our experiments we have not studied refinements of models so far but foresee this for the next steps.

2.2 Experiment 2: Business Process Diagrams

In a second experiment, we considered the modeling of Business Processes. For our initial experiments, we used a simplified subset of block-structured BPMN diagrams supporting start- and end nodes, XOR- and PAR- gateways, and control-flow edges between these elements.

In our prompt template, we described the metamodel in natural language by first defining the supported node types and then explaining the allowed control-flow edges between nodes. e.g., a task has exactly one successor and one predecessor; nodes of type XOR-Split have one predecessor node and two successor nodes (*successor_true*, *successor_false*) and a Boolean condition

After the metamodel was defined, we provided a textual process description containing sequences of activities, a condition, and the parallel execution of tasks. Therefore, the process model for the textual description requires all elements of the metamodel. In order to train ChatGPT to generate a custom simplified JSON serialization format for process models, we provided a correct model for the given textual description in JSON format. This completes the introduction of ChatGPT for the process modeling task template. The whole definition is contained in Prompt 5 in the Appendix on page 13.

We then asked ChatGPT (GPT-4) to create a JSON output for a new modeling scenario as shown in Task Definition 2.

Task Definition 2

First, a claim is filed, then the claim is checked. If the claim is valid, a reimbursement is issued. Otherwise, the claim is rejected.

The JSON output created by ChatGPT by using the GPT-4 model is shown in the Prompt Result 3 in the Appendix on page 14. Fig. 3 shows a graphical representation of the JSON output.

The output correctly models the given process description, and it did not include non-necessary elements like parallel gateways. The generated

process model is syntactically correct, and it is a valid instance of the provided metamodel.

2.3 Experiment 3: UML Class Diagrams

For experiments on UML class diagrams, we decided to put a focus on a Zero-Shot approach, i.e. by re-using an existing syntax that is known to ChatGPT. This is available for example via the PlantUML approach³, where UML diagrams and some other diagram types can be generated programmatically using a domain-specific language. In previous attempts, we had found that ChatGPT is capable of generating according models. Thus, we designed the prompt template shown in Prompt 1.

Prompt 1

Consider the following scenario and represent it as a UML class diagram in PlantUML code and do not add any information that is not described here and do not add any explanations, only show the plain JSON output using triple backticks before and after the code: \$code

We then added a problem description as shown in Task Definition 3. It is a translation of a UML exercise taken from an introductory UML book by Seidl et al. (2012).

Task Definition 3

A university consists of multiple faculties, which are composed themselves of multiple institutes. Every faculty and each institute have a description. Each institute has an address. Every faculty is led by a dean, who is an employee of the faculty. The total number of employees is known. Each employee has a social security number, a name and an email address. It is distinguished between scientific and administrative employees. Scientific employees are assigned to at least one institute. For each scientific employee, his research area is known. In addition, scientific employees can participate in projects for a particular number of hours. For the projects, the name and the start and end date are known. Some scientific employees give courses. Then they are denoted as lecturers. Courses have a unique number, a name, and a number of hours.

³ See <https://plantuml.com/en/>

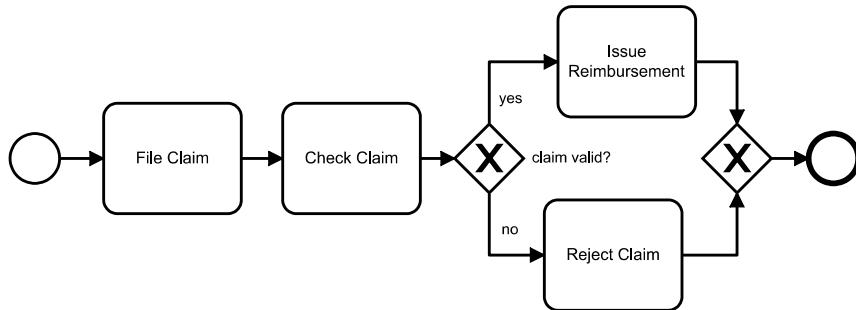


Figure 3: Generated Output for the Process Modeling task as a BPMN diagram.

The result created by ChatGPT for this prompt is shown in Fig. 4 – we do not list here the returned PlantUML code, but rather only the graphical representation that can be directly generated from it either.

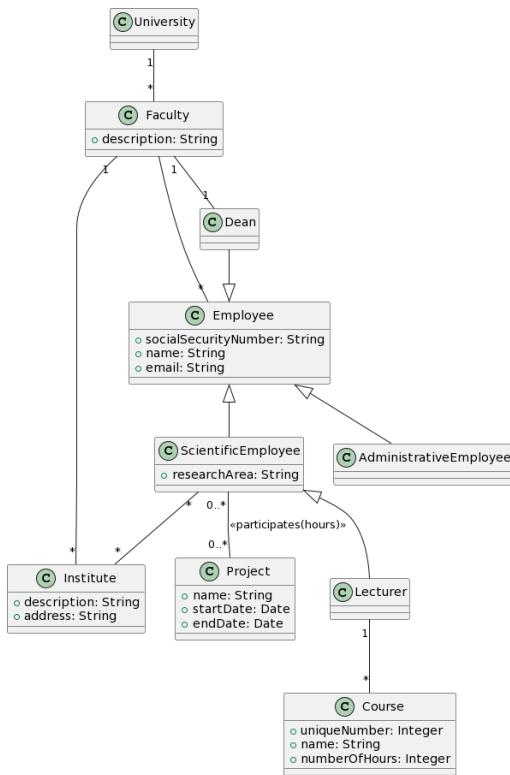


Figure 4: Result of the Prompt for Generating UML Class Diagrams in PlantUML Notation for the Sample from (Seidl et al. 2012)

In the diagram, most information is correctly represented, and we found the result quite impressive. What could be remarked is that only

the association between *Scientific Employee* and *Project* is labeled, whereas this has not been done for the other associations, but we consider this a rather minor issue. We suspected that the training of GPT may have included several types of class diagrams in PlantUML notation, which may be the reason for the good performance here. An obvious argument would thus be that this concrete example may have been learned by GPT before. Thus, we decided to conduct another experiment with a more abstract problem definition.

This additional exercise is shown in Task Definition 4. We made it up ourselves and consider it thus highly unlikely that it had been presented to GPT in the course of its training before. The result generated by ChatGPT in PlantUML notation is shown in Fig. 5. Also, this example is fully correct and corresponds perfectly to the text input.

Task Definition 4

A class A has three subclasses B, C and D. B and D both have a name and a date as attributes. B is also a subclass of G, which is related to at least one or more classes of type H. G has an attribute named X and H has an attribute named Omega.

2.4 Experiment 4: HERAKLIT Models

For the generation of HERAKLIT models we used Prompt 6 as described in the Appendix on page 15. The example, which is inserted into the prompt template, is based on a simple scenario containing a fan and a lighting in a bathroom described in Task Definition 5 in more details (Source: Fettke and Reisig 2023).

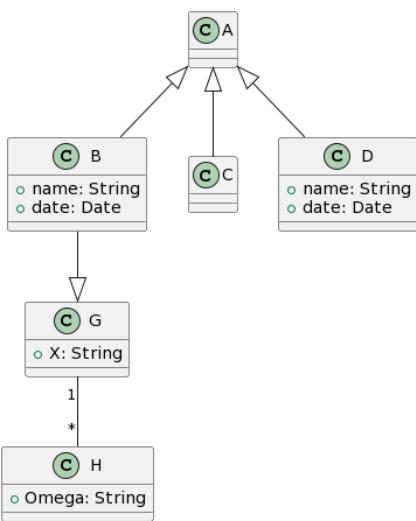


Figure 5: Result of the Prompt for Generating UML Class Diagrams in PlantUML Notation for the Abstract Problem Definition

Task Definition 5

Lighting and ventilation of a bathroom: If the light is switched on when the fan is stationary, the fan also starts after a while. Then, when the light is turned off, the fan continues to run for some time. If the light is turned on first and then turned off quickly when the fan is stationary, the fan will not start at all. If the light is first switched off and then quickly switched on again when the fan is running, the fan continues to run without interruption.

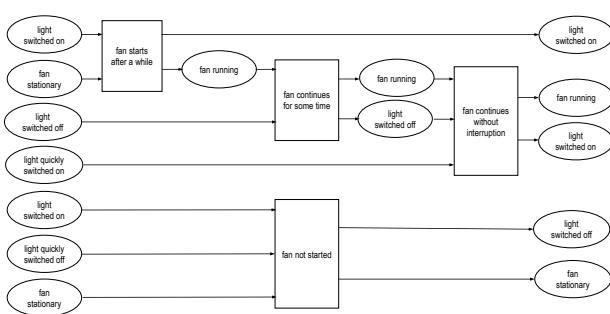


Figure 6: Result of the Prompt for Generating HERAKLIT Models based on the Task Definition 5

Fig. 6 shows the GPT-generated run. A run in HERAKLIT describes the causal order of events

of a system (Fettke and Reisig 2021, 2022). A deeper analysis shows clearly:

- The defined JSON-format can correctly be generated by GPT.
- The four generated event atoms correctly correspond to the intended scenario description.
- Generated labels for states seem in many cases reasonable, for instance, *light switched on*, *fan running*, *light switched off*, *fan stationary*. Other modeling choices may be debatable, for instance, *light quickly switched off*. Since a light switch might be turned *on* or *off*, but it is unfamiliar to identify the state of a switch with the duration of its switching.
- Generated labels for event atoms seem unusual, too. The event *fan starts after a while* seems reasonable. The event *fan not started* seems to denote a state. The two other events are similar, unusual.
- The causal order of states and events are reasonable from a first view. However, a deeper analysis shows clear limitations. Neither the light cannot be switched on and off, nor the fan starts and stops according to the generated model.

An overall evaluation of the scenario can be evaluated with the mark "sufficient": Some basic structure is acceptable, but a deeper understanding of the bathroom scenario based on this model is clearly lacking.

3 Discussion and Evaluation of the Results

Based on the experiments we conducted, large language models such as ChatGPT may act in our view as a new type of interface for humans to interact with any kind of software or hardware system. In contrast to previous interaction approaches, natural language is put at the core of such approaches and may either be entered through text or via audio and even visual means⁴. As has been recently announced by Microsoft, it is soon to be expected

⁴ See the latest reports on the multimodal interaction in future GPT versions: <https://openai.com/research/gpt-4>.

that these functionalities will be available in its Microsoft 365 product range⁵. Thus, it seems worthwhile to further explore how such an integration can be achieved for tools in conceptual modeling and in particular for enterprise modeling tasks in the future.

The evaluation of the generated results for ER, BPMN, UML, and HERAKLIT models shows that very large parts of models can be correctly generated by ChatGPT. However, despite some impressive results that are fully correct, it still requires experience in modeling to validate the results. As has been found already for other use cases of ChatGPT and similar language models, the main advantage of them can be seen in their ability to quickly generate a first draft result, which can then be refined in a stepwise fashion. In contrast to initial assumptions that ChatGPT possesses *intelligence* or *understanding* which had been frequently uttered in the media, we rather see it as a tool for supporting experts to arrive at results more quickly and easily than through pure manual interaction. This can be well illustrated with the results derived in our experiments.

Initially, we had conducted the experiments using the GPT-3 model in ChatGPT. As soon as GPT-4 became available, we switched to this model and noticed a considerable improvement of the generated results. Thus, we recommend for any further experiments or developments to use the most recent version of the language model.

4 Conclusion and Future Research

The experiments conducted so far showed the enormous potential of large language models such as ChatGPT for supporting modeling tasks, if a textual problem description of the domain to be modeled exists. We were truly impressed how well both the zero-shot as well as the few-shot approaches worked. In particular, the strict adherence to the custom formats provided in the prompts was impressive and initially unexpected.

⁵ <https://www.microsoft.com/en-us/microsoft-365/blog/2023/03/16/introducing-microsoft-365-copilot-a-whole-new-way-to-work/>

Through the experiments, we could identify a number of open research questions and potential directions for future research. We group them by *methodological considerations* for further experiments, *aspects of prompt engineering for conceptual modeling*, and *language-model related issues*.

In terms of methodological considerations for future experiments, more objective *evaluation strategies* of the generated solutions need to be foreseen. This could be done by reverting to established methods in teaching and examination of conceptual modeling, which is common at many institutions. Collected prior examination results from students could serve as a baseline for evaluating the output generated by large language models. In this context, it will need to be investigated whether the two *use cases* proposed by us – i. e. generating and interpreting models – are sufficient or whether more use cases can be imagined.

For the engineering of prompts in conceptual modeling, further research is necessary to find out which *modeling languages and notations* work best for large language models. A potential direction, which is currently being explored by some of our students, includes the use of specifically developed *domain-specific languages* for few-shot prompts, which include more complex descriptions of the formal frameworks underlying the models. Besides PlantUML also EMF has been found to be known to ChatGPT and could be used for further experiments. These approaches could lead to even better results for generating models and giving feedback to existing model solutions. For the *task definition* part of the prompts it will need to be assessed whether for example colloquial descriptions, workshop minutes, semi-formal structures or stepwise refinements work best.

Finally, in terms of the underlying large language models, it will have to be explored if the recently proposed open-source solutions can compete with the ChatGPT and OpenAI APIs⁶. These

⁶ See for example GPT4all <https://github.com/nomic-ai/gpt4all>, Alpaca <https://crfm.stanford.edu/2023/03/13/alpaca.html>.

have the benefit of permitting detailed insights into the technical architecture and are not constrained by commercial barriers, which allows for an unlimited number of experiments and easier integration into existing tools. It may even be feasible to add specific modeling-oriented training data to such models for improving the performance in terms of model generation and interpretation.

References

- Ali S. R., Dobbs T. D., Hutchings H. A., Whitaker I. S. (2023) Using ChatGPT to write patient clinic letters. In: *The Lancet Digital Health*, E179–E181
- Brown T. B., Mann B., Ryder N., Subbiah M., Kaplan J., Dhariwal P., Neelakantan A., Shyam P., Sastry G., Askell A., Agarwal S., Herbert-Voss A., Krueger G., Henighan T., Child R., Ramesh A., Ziegler D. M., Wu J., Winter C., Hesse C., Chen M., Sigler E., Litwin M., Gray S., Chess B., Clark J., Berner C., McCandlish S., Radford A., Sutskever I., Amodei D. (2020) Language Models are Few-Shot Learners. In: (arXiv:2005.14165)
- Campbell M. (2020) Beyond Conversational Artificial Intelligence. In: *Computer* 53(12), pp. 121–125
- Choi J. H., Hickman K. E., Monahan A., Schwarcz D. B. (2023) ChatGPT Goes to Law School. In: SSRN Minnesota Legal Studies Research Paper (No. 23-03)
- Dwivedi Y. K., Kshetri N., Hughes L., Slade E. L., Jeyaraj A., Kar A. K., Baabdullah A. M., Koohang A., Raghavan V., Ahuja M., Albanna H., Albashrawi M. A., Al-Busaidi A. S., Balakrishnan J., Barlette Y., Basu S., Bose I., Brooks L., Buhalis D., Carter L., Chowdhury S., Crick T., Cunningham S. W., Davies G. H., Davison R. M., Dé R., Dennehy D., Duan Y., Dubey R., Dwivedi R., Edwards J. S., Flavián C., Gauld R., Grover V., Hu M.-C., Janssen M., Jones P., Junglas I., Khorana S., Kraus S., Larsen K. R., Latreille P., Laumer S., Malik F. T., Mardani A., Mariani M., Mithas S., Mogaji E., Nord J. H., O'Connor S., Okumus F., Pagani M., Pandey N., Papagiannidis S., Pappas I. O., Pathak N., Pries-Heje J., Raman R., Rana N. P., Rehm S.-V., Ribeiro-Navarrete S., Richter A., Rowe F., Sarker S., Stahl B. C., Tiwari M. K., van der Aalst W., Venkatesh V., Viglia G., Wade M., Walton P., Wirtz J., Wright R. (2023) “So what if ChatGPT wrote it?” Multidisciplinary perspectives on opportunities, challenges and implications of generative conversational AI for research, practice and policy. In: *International Journal of Information Management* 71, p. 102642
- Fettke P., Reisig W. (2021) Handbook of HERAKLIT, HERAKLIT working paper, v1.1, September 20, 2021, <http://www.heraklit.org>
- Fettke P., Reisig W. (2022) Breathing Life into Models: The Next Generation of Enterprise Modeling. In: Hans-Georg Fill Marten van Sinderen L. M. (ed.) *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*. SCITEPRESS, pp. 7–14
- Fettke P., Reisig W. (2023) Understanding Computer-Integrated Systems – How to model the digital world with HERAKLIT. in preparation
- GitHub (2021) GitHub Copilot Your AI pair programmer. <https://copilot.github.com/>
- Jurafsky D., Martin J. H. (2023) *Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, Third Edition Draft. Draft
- Nguyen N., Nadi S. (2022) An Empirical Evaluation of GitHub Copilot’s Code Suggestions. In: *Proceedings of the 19th International Conference on Mining Software Repositories. MSR ’22*. ACM, Pittsburgh, Pennsylvania, pp. 1–5
- Ouyang L., Wu J., Jiang X., Almeida D., Wainwright C. L., Mishkin P., Zhang C., Agarwal S., Slama K., Ray A., Schulman J., Hilton J., Kelton F., Miller L., Simens M., Askell A., Welinder P., Christiano P., Leike J., Lowe R. (2022) Training language models to follow instructions with human feedback. In: (arXiv:2203.02155)

Pearce H., Ahmad B., Tan B., Dolan-Gavitt B., Karri R. (2022) Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 754–768

Ren R., Castro J. W., Santos A., Pérez-Soler S., Acuña S. T., de Lara J. (2020) Collaborative Modelling: Chatbots or On-Line Tools? An Experimental Study. In: Li J., Jaccheri L., Dingsøyr T., Chitchyan R. (eds.) EASE '20: Evaluation and Assessment in Software Engineering, Trondheim, Norway, April 15-17, 2020. ACM, pp. 260–269

Seidl M., Brandsteidl M., Huemer C., Kappel G. (2012) UML @ Classroom - Eine Einführung in die objektorientierte Modellierung. dpunkt.verlag

Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L., Polosukhin I. (2017) Attention is All you Need. In: Guyon I., Luxburg U. v., Bengio S., Wallach H. M., Fergus R., Vishwanathan S. V. N., Garnett R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pp. 5998–6008

Weizenbaum J. (1966) ELIZA - a computer program for the study of natural language communication between man and machine. In: Communications of the ACM 9(1), pp. 36–45

Weng L. (2023) Prompt Engineering <https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/>

White J., Hays S., Fu Q., Spencer-Smith J., Schmidt D. C. (2023) ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design

Zhao T. Z., Wallace E., Feng S., Klein D., Singh S. (2021) Calibrate Before Use: Improving Few-Shot Performance of Language Models

Appendix: Experimental methodology

This appendix documents the used experimental methodology.

A Overall Architecture

Our approach builds on the idea of GraphGPT. The approach permits to alter the thus generated graphs, e. g. by instructing ChatGPT to color certain nodes. The approach builds upon a simple web application with an interface to the ChatGPT API. Thereby, a prompt is pre-defined that describes the context for ChatGPT to make it understand in which way such graphs are to be derived from a given text – see Prompt 2. Any text inserted by the user is then embedded in this prompt, together with the previously generated graph. Due to the input limitation of ChatGPT, the approach does however not seem to be indefinitely scalable as the input is currently limited to 4096 tokens⁷. We can identify in the prompt definition of GraphGPT that at first some context needs to be handed over to the GPT API for explaining the concept of graphs and how they are to be represented in a simplified JSON code. This also includes the definition of an initial state of the graph and an explicit example on how the graph is to be modified upon some prompt given by the user. In the simplified representations of graphs, only arrays for nodes and edges are given, including identifiers, labels and colors.

Prompt 2

Given the current state of a graph and a prompt, extrapolate as many relationships as possible from the prompt and update the state. Every node has an id, label, and color (in hex). Every edge has a to and from with node ids, and a label. Edges are directed, so the order of the from and to is important.

Examples:

```
current state:  
{ "nodes": [ { "id": 1, "label": "Bob",  
"color": "#ffffff" } ], "edges": [] }
```

```
prompt: Alice is Bob's roommate. Make her node  
green.
```

```
new state:  
{ "nodes": [ { "id": 1, "label": "Bob",  
"color": "#ffffff" }, { "id": 2,  
"label": "Alice", "color": "#00ff00" } ],  
"edges": [ { "from": 1, "to": 2, "label": "is roomate" } ] }
```

⁷ Tokens used by GPT-3 and GPT-4 are encoded and decoded using Byte-Pair-Encoding (BPE) – see <https://huggingface.co/course/chapter6/5?fw=pt>. A tool for calculating the number of tokens in a text is provided here: <https://platform.openai.com/tokenizer>

```

"label": "Alice", "color": "#ff7675" } ],
"edges": [ { "from": 1, "to": 2,
"label": "roommate" } ] }

current state:
$state

prompt: $prompt

new state:

```

In terms of the underlying technical architecture, the approach as shown by GraphGPT gives hints on the necessary components. As shown in Fig. 7 for the case of natural language input, the input given by a user is inserted by an application into a pre-defined prompt structure. This pre-defined prompt thereby acts as a kind of template. The result of this insertion is then sent as the actual prompt, i. e. the input for a large language model, to the GPT API. The result of the GPT API is retrieved by the software application and further processed. This processing may include any kind of algorithmic procedure, e. g. for transforming the result into different representations. As shown by the approach of GraphGPT, the result or parts thereof may be inserted back into subsequent prompt definitions, thus acting as variable input or state transfer.

What has to be noted in this kind of architecture is that there are no specific components necessary for pre-processing the natural language input by the user, nor for any kind of context services apart from the context description embedded in the prompt. Any kind of format specification thus has to be embedded in the prompt and needs to be explained to the language model so that it can embed it in the generation of the result. This format specification can be given in the form of examples or may refer to some format that is known to ChatGPT, e. g. as we will show later for generating PlantUML code. The result is thereby not deterministic, and we discovered throughout our experiments that sometimes unexpected variations occur in the output. Such variations may be corrected by subsequent prompts, by asking the language model to perform some modification, but it is currently unclear to us how such

variations can be effectively controlled, especially when there is some kind of software interface that expects a particular output. Further experiments would have to inspect this behavior in more detail.

B Prompt Engineering for Conceptual Modeling

For our experiments we considered two possible directions: a. the generation of conceptual models via ChatGPT and b. the interpretation of models. Building upon the ideas of GraphGPT, we used the following prompt structure, as shown in Fig. 8.

At the beginning, typically some *context explanation* needs to be specified. This corresponds to the approach of *few-shot learning*, which has been described for letting large language models learn specific tasks from a few examples (Zhao et al. 2021). Another strategy would be to use *zero-shot learning*, where no examples are presented, and the model is asked to generate the output directly. In the case of conceptual models, this context presentation may include a simplified characterization of the metamodel of the modeling language, potentially enriched with an example. Due to the limitations in the number of input tokens, it is currently not possible to hand over complete metamodel specifications at this stage. A future approach could include to train a language model explicitly on metamodels, so that it already incorporates these concepts. For some existing modeling languages such metamodels are already included in GPT, e. g. for the case of PlantUML-encoded models, which is an executable textual notation for generating graphical representations of models in different modeling languages. After the context explanation, a *state description* can be foreseen, e. g. for re-inserting an already existing model output. Subsequently, *output instructions* are handed over to instruct the language model about the styling of the output and force it for example to omit any explanations of the output, which it may otherwise generate. This step of *instruction prompting* is essential to permit a direct processing of the retrieved results by a software

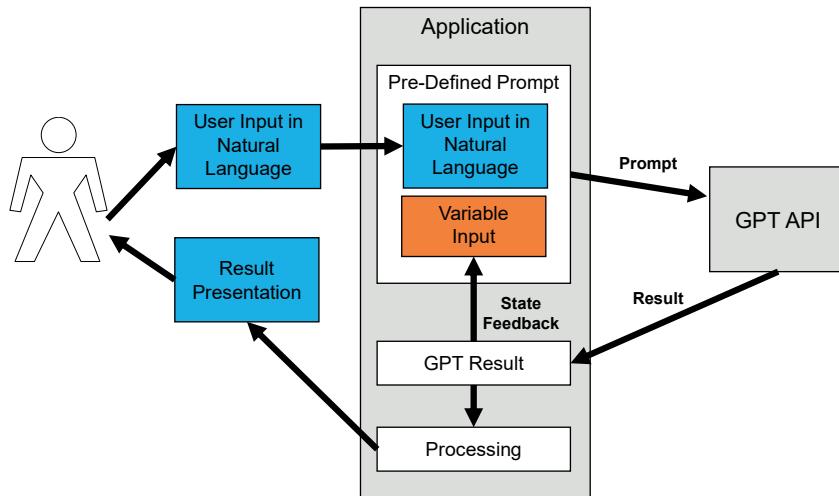


Figure 7: General Architecture for Natural Language Input as Derived from GraphGPT

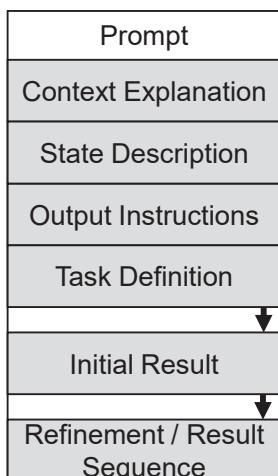


Figure 8: Prompt Structure Used for the Experiments

application that expects a particular format⁸. Finally, the actual *task definition* is appended, which may be dynamically inserted based on user input. The step's context explanation, state description, and output instructions would correspond to the *pre-defined prompt* as described above and the problem definition to the *user input in natural language*.

After this input has been sent to the language model, some *initial result* is retrieved. Subsequent interactions with the language model can then build upon this result, either by accessing the same session state of the last output – as in the case of the web-based ChatGPT application, or by re-inserting the state into the *state description*. Thereby, a sequence of *refinements and results* can follow.

C ER Diagram Prompts and Results

For the generation of ER diagrams we used the following prompt template:

Prompt 3

Assume the following for the definition of ER diagrams: An ER diagram consists of the following nodes: Entity Types, Relationship Types, and Attributes. Each node has a name. A Relationship Type is always connected to two Entity Types using the "assign" edge. Each of these "assign" edges has a cardinality to express how many entity types participate in a relationship type. The cardinality must have one of the following values indicating the minimum and the maximum (0,1), (1,1), (0,N), (1,N) whereby the N stands for an unlimited number of entities. An "assign" edge is formally specified as "EntityType, Relationship Type, Cardinality". Ensure that every Relationship Type has at least two "assign" edges relating to it. Attributes can be attached to Entity Types and Relationship Types via an Edge named "has_attribute".

⁸ For further hints and experiences on prompt engineering see (Weng 2023).

Here is an example for an ER diagram in JSON syntax, which expresses that Persons own no or several cars and that cars may be parked in one garage. Further that cars always have to be owned by at least one person:

```
{"ERdiagram": [
    "EntityTypes": [
        "Person", "Car", "Garage"
    ],
    "RelationshipTypes": [
        "owns", "parked_in"
    ],
    "Attributes": [
        "Name", "Numberplate", "GarageNo"
    ],
    "assign": [
        "Person, owns, (0,N)",
        "Car, owns, (1,N)",
        "Car, parked_in, (0,1)",
        "Garage, parked_in, (0,N)"
    ],
    "has_attribute": [
        "Person, Name",
        "Car, Numberplate",
        "Garage, GarageNo"
    ]
}]}
```

Create a new ER diagram using this syntax for the following scenario and do not add any information that is not described here and do not add any explanations, only show the plain JSON output using triple backticks before and after the code: \$code

The resulting output that is generated by ChatGPT and GPT-4 is then as follows:

Prompt Result 2

```
{ "ERdiagram": {
    "EntityTypes": [
        "Hotel",
        "Room",
        "Guest"
    ],
    "RelationshipTypes": [
        "has",
        "part_of",
        "books"
    ],
    "Attributes": [
        "RoomNumber"
    ],
    "assign": [
        "Hotel, has, (1,N)",
        "Room, part_of, (1,1)",
        "Guest, books, (0,N)",
        "Room, books, (0,N)"
    ],
    "has_attribute": [
        "Room, RoomNumber"
    ]
}}
```

```
    ]
}
}
```

For the interpretation of ER diagrams, we used the following prompt template:

Prompt 4

Assume the following for the definition of ER diagrams: An ER diagram consists of the following nodes: Entity Types, Relationship Types, and Attributes. Each node has a name. A Relationship Type is always connected to two Entity Types using the "assign" edge. Each of these "assign" edges has a cardinality to express how many entity types participate in a relationship type. The cardinality must have one of the following values indicating the minimum and the maximum (0,1), (1,1), (0,N), (1,N) whereby the N stands for an unlimited number of entities. An "assign" edge is formally specified as "EntityType, Relationship Type, Cardinality". Ensure that every Relationship Type has at least two "assign" edges relating to it. Attributes can be attached to Entity Types and Relationship Types via an Edge named "has_attribute". Here is an example for an ER diagram in JSON syntax, which expresses that Persons own no or several cars and that cars may be parked in one garage. Further, that cars always have to be owned by at least one person:

```
{"ERdiagram": [
    "EntityTypes": [
        "Person", "Car", "Garage"
    ],
    "RelationshipTypes": [
        "owns", "parked_in"
    ],
    "Attributes": [
        "Name", "Numberplate", "GarageNo"
    ],
    "assign": [
        "Person, owns, (0,N)",
        "Car, owns, (1,N)",
        "Car, parked_in, (0,1)",
        "Garage, parked_in, (0,N)"
    ],
    "has_attribute": [
        "Person, Name",
        "Car, Numberplate",
        "Garage, GarageNo"
    ]
}]}
```

Interpret now the following ER diagram given in the notation from above and describe its content textually: \$code

The diagram in Fig. 2 was then handed over to ChatGPT in the following custom notation:

Task Definition 6

```
{
  "ERdiagram": [
    "EntityTypes": [
      "University", "Teacher", "Course"
    ],
    "RelationshipTypes": [
      "teaches", "belongs_to"
    ],
    "Attributes": [
      "Name"
    ],
    "assign": [
      "University, belongs_to, (0,N)",
      "Teacher, belongs_to, (1,N)",
      "Teacher, teaches, (0,N)",
      "Course, teaches, (1,N)"
    ],
    "has_attribute": [
      "University, Name",
      "Teacher, Name",
      "Student, Name"
    ]
  ]
}
```

D Workflow Prompts and Results

The following prompt template was used for the workflow experiments:

Prompt 5

Assume the following for the definition of Workflow Models: A Workflow Model consists of the following node types: Task, XOR-Split, XOR-Join, PAR-Split, PAR-Join, Start-Node, End-Node. Each node of type Task has a label. Nodes of type Start-Node have exactly one successor node and no predecessor node. Nodes of type End-Node have exactly one predecessor node and no successor node. Nodes of type Task have exactly one successor node and exactly one predecessor node. Nodes of type XOR-Split have one predecessor node and two successor nodes (successor_true, successor_false) and a boolean condition. The successor_true is executed if the condition is true, and the successor_false is executed if the condition is false. Nodes of type XOR-Join have two predecessor nodes (predecessor_true, predecessor_false) and one successor node. The successor gets active after one predecessor is executed. Nodes of type PAR-Split are used to support the parallel execution of tasks. They have one predecessor and two successor nodes (successor_A, successor_B). The successor nodes are executed in parallel. Nodes of type PAR-Join have two predecessor nodes

(predecessor_A, predecessor_B) and one successor node. The successor node is activated once both predecessors are executed. All node types except for start-node and end-node are optional. If a process contains a xor-split node, it must also contain a xor-join node. If a process contains a par-split node, it must also contain a par-join node. Here is an example of a workflow model in JSON syntax, which represents that first, the start node is activated, then the task "Process Order" is executed. Then the task "Check Stock" is executed. Then an XOR-Split Gateway is executed. Depending on the condition "items available?" either the PAR-Split node is activated, and the two tasks, "Ship Items" and "Confirm Order," are executed in parallel, or if the items are not in stock, the task "Reject Order" is executed. Finally, the end-node is activated:

```
{
  "WorkflowModel": {
    "start-Node": {
      "id": "S1",
    },
    "end-node": {
      "id": "E1",
    },
    "tasks": [
      {
        "id": "T1",
        "label": "Process Order",
        "successor": "T2",
        "predecessor": "E1"
      },
      {
        "id": "T2",
        "label": "Check Stock",
        "successor": "XS1",
        "predecessor": "T1"
      },
      {
        "id": "T3",
        "label": "Ship Items",
        "successor": "PJ1",
        "predecessor": "PS1"
      },
      {
        "id": "T4",
        "label": "Confirm Order",
        "successor": "PJ1",
        "predecessor": "PS1"
      },
      {
        "id": "T5",
        "label": "Reject Order",
        "successor": "XJ1",
        "predecessor": "XS1"
      }
    ],
    "gateways": [
      {
        "type": "XOR-Split",
        "id": "XS1"
      }
    ]
  }
}
```

```

    "id" : "XS1",
    "condition" : "items available?",
    "successor_true" : "PS1",
    "successor_false" : "T5",
    "predecessor" : "T2"
},
{
  "type" : "XOR-Join",
  "id" : "XJ1",
  "predecessor_true" : "PS1",
  "predecessor_false" : "T5",
  "successor" : "E1"
},
{
  "type" : "PAR-Split",
  "id" : "PS1",
  "successor_A" : "PS1",
  "successor_B" : "T5",
  "predecessor" : "T2",
},
{
  "type" : "PAR-Join",
  "id" : "PJ1",
  "predecessor_A" : "T3",
  "predecessor_B" : "T4",
  "successor" : "XJ1"
}
]
}
}

```

Create a new Workflow Model using this syntax for the following scenario and do not add any information that is not described here and do not add any explanations, only show the plain JSON output using triple backticks before and after the code: \$code

The result for the input of the Task Definition 2 is generated in the custom JSON format as follows:

Prompt Result 3

```
{
  "WorkflowModel": {
    "start-Node": {
      "id": "S1"
    },
    "end-node": {
      "id": "E1"
    },
    "tasks": [
      {
        "id": "T1",
        "label": "File Claim",
        "successor": "T2",
        "predecessor": "S1"
      },
      {
        "id": "T2",
        "label": "Check Claim",
        "successor": "XS1",
      }
    ]
  }
}
```

```

    "predecessor": "T1"
},
{
  "id": "T3",
  "label": "Issue Reimbursement",
  "successor": "XJ1",
  "predecessor": "XS1"
},
{
  "id": "T4",
  "label": "Reject Claim",
  "successor": "XJ1",
  "predecessor": "XS1"
],
"gateways": [
  {
    "type": "XOR-Split",
    "id": "XS1",
    "condition": "claim valid?",
    "successor_true": "T3",
    "successor_false": "T4",
    "predecessor": "T2"
  },
  {
    "type": "XOR-Join",
    "id": "XJ1",
    "predecessor_true": "T3",
    "predecessor_false": "T4",
    "successor": "E1"
  }
]
}

```

E HERAKLIT Prompts and Results

For the experiments for generating HERAKLIT models, we used Prompt 6.

Prompt 6

Assume the following for HERAKLIT models: A Model consists of "net-atoms". Each net-atom has a "left interface", a "transition" and a "right interface". Consider the following example for a HERAKLIT model:

```
{
  "run": [
  {
    "net-atoms": [
      {
        "left-interface" :
          "vendor V2 available",
        "transition" :
          "shoes to be ordered",
        "right-interface" :
          "vendor V2 available"
      },
      {
        "left-interface" :
          "shoes to be ordered",
        "transition" :
          "shoes to be delivered",
        "right-interface" :
          "customer V2 available"
      }
    ]
  }
]
```

```

        "left-interface" :
          "vendor V2 available",
          "Claire wants a hat",
        "transition" :
          "hat not on offer",
        "right-interface" :
          "vendor V2 available",
          "Claire without hat"
      },
    ],
  }
}

Create a new HERAKLIT model using this syntax
for the following scenario and do not add any
information that is not described here and do
not add any explanations, only show the plain
JSON output using triple backticks before and
after the code: $code

```

```

"transition": "fan not started",
"right-interface": [
  "light switched off",
  "fan stationary"
],
{
  "left-interface": [
    "light switched off",
    "light quickly switched on",
    "fan running"
  ],
  "transition":
    "fan continues without
    interruption", "right-interface": [
    "light switched on",
    "fan running"
  ]
}
]
}

```

The result generated by ChatGPT for the Prompt 6 for HERAKLIT models together with Task Definition 5 led to Prompt Result 4.

Prompt Result 4

```
{
  "run": [
    {
      "net-atoms": [
        {
          "left-interface": [
            "light switched on",
            "fan stationary"
          ],
          "transition": "fan starts after
            a while",
          "right-interface": [
            "light switched on",
            "fan running"
          ]
        },
        {
          "left-interface": [
            "light switched off",
            "fan running"
          ],
          "transition": "fan continues for
            some time",
          "right-interface": [
            "light switched off",
            "fan running"
          ]
        },
        {
          "left-interface": [
            "light switched on",
            "light quickly switched off",
            "fan stationary"
          ],
          "transition": "fan continues without
            interruption", "right-interface": [
            "light switched on",
            "fan running"
          ]
        }
      ]
    }
  ]
}
```