

Exploiting Large Language Models for the Automated Generation of Constraint Satisfaction Problems

Lothar Hotz^{1,*†}, Christian Bähnisch^{1,†}, Sebastian Lubos², Alexander Felfernig²,
Albert Haag³ and Johannes Twiefel⁴

¹Hamburger Informatik Technologie-Center e.V., Hamburg, Germany

²Graz University of Technology, Graz, Austria

³Product Management Haag GmbH, Bad Dürkheim, Germany

⁴exXa GmbH, Hamburg, Germany

Abstract

Constraint Satisfaction Problems (CSPs) are a core technology that solves many real-world problems, especially for configuration tasks. A key success factor in this context is an efficient knowledge acquisition process where domain experts and knowledge engineers (developers of CSPs) should develop an agreement on the correctness of the expanding knowledge base as soon as possible. In this paper, we show how large language models (LLMs) can be applied to the automated generation of solutions for constraint satisfaction problems thus reducing overheads related to CSP development and maintenance in the future.

Keywords

Constraint Satisfaction Problems, Large Language Models, Knowledge Acquisition, Automated Generation

1. Introduction

Knowledge acquisition for knowledge-based systems, especially constraint-based systems, is a complex task. It includes the formalization of partly tangible knowledge with a knowledge model, such as a configuration, constraint, or feature model [1, 2]. Constrained-based systems, in particular, are often used to implement configuration systems, due to their ability to compute possible values for configurations or even directly configurations themselves [3]. Hence, modeling a configuration problem as a constraint satisfaction problem (CSP) is a typical approach for computing configurations. This type of modeling is similar to developing programs in a high-level programming language but also incorporates logical semantic elements.

Currently in software engineering, the use of large language models (LLMs) is exploited to support programmers in their daily tasks, such as coding, code completion, reviewing code, API programming, generating test cases, documentation, or identifying

design patterns as well as learning programming languages or understanding legacy code [4]. This is achieved through general LLMs such as ChatGPT¹ or specific ones like for coding trained LLMs such as CoPilot², or software agents like CREWAI³ and AutoGenStudio⁴, or software assisting engineers such as Devine AI⁵ or its open source correspondent Devika⁶.

An example of developing a small game with a graphical user interface is GPT-Engineer⁷, which prompts in natural languages and provides the game logic and the graphical interface through HTML and CSS coding. Especially the combination of an LLM with a compiler or interpreter in a chain leads to a self-evaluation where a code, which is generated code by an LLM, is directly checked through the interpreter, and the resulting error message is taken as the next prompt for the LLM. Through iterating this (e.g., with a supporting tool such as LangCHAIN⁸) a syntactically correct program is created. However, broader tasks, such as reviewing a software architecture or refactoring code distributed over multiple files are hindered by the limited size of a

ConfWS'24: 26th International Workshop on Configuration, Sep 2–3, 2024, Girona, Spain

*Corresponding author.

[†]These authors contributed equally.

✉ lothar.hotz@uni-hamburg.de (L. Hotz);
christian.baehnisch@uni-hamburg.de (C. Bähnisch);
slubos@ist.tugraz.at (S. Lubos); afelfern@ist.tugraz.at
(A. Felfernig); albert@product-management-haag.de
(A. Haag); twiefel@exxxa.ai (J. Twiefel)

0000-0001-7370-7726 (L. Hotz)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

¹Chat Generative Pre-Trained Transformer

²<https://github.com/features/copilot>. All URLs accessed in May 2024

³<https://www.crewai.com/>

⁴<https://microsoft.github.io/autogen/blog/2023/12/01/AutoGenStudio>

⁵<https://preview.devin.ai/>

⁶<https://github.com/stitionai/devika>

⁷<https://github.com/gpt-engineer-org/gpt-engineer>

⁸https://python.langchain.com/docs/get_started/introduction

prompt (the context window) to a given number of tokens such as 128,000 input tokens for GPT-4. In summary, applying LLMs to software development still has to cope with issues such as limited context windows, maintainability of AI-created code, and applicability in project planning or software architecture.

Inspired by these approaches, for constraint satisfaction problems the following questions arise:

- Which knowledge acquisition and engineering tasks can be supported by LLMs?
- Can knowledge models be generated by LLMs? How is quality ensured?
- Which alternative implementations of a knowledge model are the most comprehensible or maintainable?

However, a main ingredient of LLMs is of course a prompt that triggers the generation of text, here of knowledge models. Hence, for a knowledge model, the questions are also:

- What type of prompts do we need?
- What has to be presented to the LLM?
- Is natural language text describing the problem enough?
- Should examples of the formal modeling language, which shall be used for formalization, be given?
- Or the complete language specification?

This paper explores the innovative use of LLMs to automatically generate constraint models in constraint programming languages like PyChoco⁹. Constraint programming involves defining problems in terms of variables, related variable domains, and constraints that must be satisfied. Popular problems in this domain include, e.g., the N-queens problem, Magic Square, Map Coloring, and the Traveling Salesperson Problem (TSP). To tackle such problems, we propose a generalized approach (Section 3) where an LLM is leveraged to generate a constraint model based on a provided prompt.

We test our approach based on several constraint satisfaction problem examples (Section 4). In this paper, we mainly concentrate on known examples from the constraint community given by well-known descriptions, not that much on new examples formulated in natural language. Although the task of starting with known problems is untypical for configuration problems, in our view, this step is necessary for exploring the possibilities of LLMs for

knowledge modeling in general. This approach of using commonly known constraint and configuration problems builds a baseline for further work.

With our proposed approach, we aim to streamline the process of model creation in constraint programming, making it faster and more accessible by harnessing the capabilities of LLMs. The main idea is to define a general prompt once which describes the problem context and then reuse this general prompt for specific knowledge engineering tasks at hand.

The remainder of this paper is organized as follows. Section 2 provides an overview of the state-of-the-art in LLM-based knowledge modeling. In Section 3, we sketch our proposed LLM-based CSP generation approach. In Section 4, we provide details regarding the LLM prompting approach based on different example CSP tasks. A discussion (Section 5) and a conclusion (Section 6) ends the paper.

2. State of the Art in LLMs and Knowledge Modeling

Many real-world problems can be interpreted as constraint satisfaction problems (CSPs). Knowledge engineers have the task of formalizing domain-specific constraints into corresponding constraint-based representations. This formalization step is often effortful and more assistance and automation are required in the modeling process which can be regarded as a major challenge for constraint-based systems [5]. Due to recent developments in the area of large language models (LLMs), we could envision agent-based approaches that support the (semi-)automated generation of CSPs [6]. In the following, we discuss related work on the application of LLMs to knowledge modeling.

The idea of exploiting synergy potentials of knowledge-based systems and language models for knowledge-base generation has been proposed a.o. by Petroni et al. [7], Ding et al. [8], and Razniewski et al. [9]. Suchanek and Luu [10] motivate the integration of LLMs with data-driven and knowledge-based approaches to exploit the advantages of both worlds, specifically, to let data-driven approaches act as a basis for grounding the LLM output in reality. In the work presented in our paper, a simple form of grounding is the usage of test cases to assure intended CSP semantics. A similar line of research is presented in Nayak and Timmapathini [11] where object and relation identification based on LLMs is investigated.

Ahmed and Choudhury [12] introduce the idea of applying LLMs to the generation of optimiza-

⁹<https://pypi.org/project/pychoco/>

tion problems. The motivation behind this is to make related problem formulation tasks more accessible to domain experts, i.e., to decrease the need for specialized mathematical knowledge to make problem formulations feasible. Based on a given dataset comprising different example problem definitions (and solutions), LLMs are used to generate optimization problem definitions for new problems defined in a textual fashion. A major difference compared to our work is that model fine-tuning is primarily based on a "human in the loop" approach whereas we focus more on automated fine-tuning giving LLM feedback on the syntactical (is the generated CSP correct or what were the compilation issues?) and the semantic level with test cases. A related approach is the application of LLMs to support different types of strategic reasoning tasks, for example, in the context of economic simulations and game theory [13].

A very similar approach provides [14]. This 'Program of Thoughts' named approach also combines an LLM with an external Python interpreter, however, in [14] for computing numerical problems. Furthermore, the verification of the generated Python program is not discussed.

The adaptation of LLMs for logical reasoning tasks [15] can be performed either based on LLM fine-tuning where LLMs are trained (and adapted) for specific problem settings [16] or based on (automated) prompt adaptation [17]. Intending to improve the quality of code generation, Pan et al. [15] follow the idea of using constraint (and SAT) solver execution errors as feedback channels (in the prompting process) to increase the quality of code generation. The feedback approach presented in our paper extends the work of Pan et al. [15] in terms of additionally providing test cases that help to improve the quality of the generated CSP also on the semantic level.

In [18], various prompts are used to create source code with built-in variability. Specific prompts enable the generation of code in different programming languages, allowing for application configuration. However, a human developer would execute these prompts, no iterative process ensures the correctness of the resulting code.

The automated generation of CSPs can also be regarded as a specific type of ontology construction where concept hierarchies are derived based on different LLM prompts representing concept hierarchy-specific queries (see, for example, the work of Funk et al. [19]). An example of such a query is the following: *What are all of the most important sub-categories of the category A?*

3. Methodology

Our approach for generating constraint models using LLMs is a structured, iterative process designed to create syntactically and semantically (almost) correct models. This begins by defining a *general LLM prompt* that can encompass the varying specifics of different constraint problems. For instance, the prompt can be tailored to a specific constraint problem but always solicits a PyChoco interpreter that solves the problem.

The process, furthermore, involves using an iterative loop where the LLM generates a constraint model based on the provided prompt. This generated model is then verified through a constraint interpreter or compiler. If the syntax of the model is incorrect, the result of the Python interpreter call is automatically taken as a prompt for a next iteration and the model is adjusted by the LLM and the process repeats. This loop continues until the syntax is correct.

Once a syntactically correct model is produced, the next step is to verify its semantics. This is done by running a series of test cases specific to the problem domain. Successful completion of test cases indicates a correct semantic implementation (at least concerning the test cases).

This approach, thus, executes an iterative prompting of an LLM, where only a human user provides the first prompt, while the results of a Python interpreter are automatically used by the LLM as successive prompts. A complete example of such an iterative dialog including in-between generated answers of the LLM ("Assistant") is presented in the Appendix Section 2.

This methodology can be further detailed through the following steps:

Leveraging Existing Knowledge Problems: To create useful constraint models, we use logical problems that are well understood and documented on the internet. Examples include classic problems such as the N-queens problem, Map Coloring, or the Traveling Salesperson Problem. These problems are selected due to their well-known constraints and solution strategies.

Integration with Constraint Solvers: Once the logical problems are defined, we connect these models to a constraint system, here PyChoco - other suitable solvers could be used. This involves mapping the logical problem's constraints to the syntax and structures of the chosen constraint language and, thus, facilitating automated solving.

Syntax Verification Loop: An iterative process is set up where the LLM generates a constraint model based on the initial prompt. This model is then

tool Python_REPL, description: A Python shell. Use this to run and verify your solution to the constraint problem. Input should be a valid Python script using pchoco. The last line should be "verify_solutions(solutions, ...)" in order to verify your solutions

System:

You are an agent designed to solve constraint problems, you write python code to solve and verify constraint problems. You have access to a python REPL, which you can use to execute python code and verify your solution. If you get an error, debug your code and try again. You only use the constraint solver pchoco. You conclude your python code with a variable "solutions" and verify it with the method call "verify_solutions(solutions, <problem_name>", where <problem_name> is given to you as the first word of the instructions. "solutions" should be a list of all solutions to the constraint problem. If verification fails, you correct your solution.

Use the following example as a template for solving constraint problems with pchoco:

User: ints10. find numbers one to 10.

Assistant:

```
'''python
from pchoco.model import Model
from test_tool import verify_solutions

model = Model()
x = model.intvar(1, 10, "x")

solver = model.get_solver()
solutions = [s.get_int_val(x) for s in solver.find_all_solutions()]
verify_solutions(solutions, "ints10")
'''
```

Figure 1: General prompt including an example for the LLM on how to call the verification method. The first three lines (the tool description) are needed to avoid iterations related to wrongly calling the method `verify_solutions` (see Appendix Section 7 for created errors without those lines).

verified for syntactic correctness using the constraint interpreter or compiler. If errors are identified, the prompt is automatically adjusted by the LLM, by using an error message as a next prompt and the LLM regenerates the model. This loop continues until a syntactically correct model is achieved.

Semantic Verification with Test Cases: After achieving a syntactically correct model, the focus shifts to ensure semantic correctness. This involves introducing test cases specific to the defined problem. The LLM-generated model runs through these test cases to verify that it not only adheres to syntactic rules but also correctly implements the problem’s logic. Iteration continues until the model passes all semantic verification tests.

Generalizability: The ultimate goal of our approach is to establish a generalizable method for generating constraint models using LLMs. By systematically using an iterative loop for both syntactic and semantic verification that employs examples and logical problem definitions, we aim to create a robust framework that can be adapted to various

constraint languages and problems.

By following these steps, our approach leverages the capabilities of LLMs to create accurate and efficient constraint models, providing a scalable solution for different problem types.

4. Prompting Constraint Models

We divide prompts into a *general part* which is used with each constraint problem, i.e., the *system prompt* (see Section 4.1), and a *query part* which points to a specific problem (see Section 4.2).

4.1. General Prompts

Prompting Syntactic Valid Models Figure 1 shows the general prompt which we developed for querying for constraint solutions. The prompt provides the following explanations to the LLM:

1. what role it has (“You are an agent...”),
2. the explanation of the REPL¹⁰ as a tool (“You have access...”),

¹⁰Read-Eval-Print-Loop

You are an agent designed to solve constraint problems, you write and execute python code to answer questions. You have access to a python REPL, which you can use to execute python code. The python REPL will keep its state between usage. Use this to gradually approach a final solution. Divide the solution into meaningful parts run them part after part verifying that each part runs correctly. If you get an error, debug your code and try again only the last part that failed. You only use the constraint solver pycoco. Only use the output of your code to answer the question. Write for each code line a comment as needed to justify your reasoning for that code line. I will give you an example of good python code, please, follow the code convention of the *good python code example*. This *good python code example* is only an example. This *good python code example* has nothing to do with the constraint problem solution. You might know the answer without running any code, but you should still run the code to get the answer. If it does not seem like you can write code to answer the question, just return "I don't know" as the answer. The user may ask questions or give follow up instructions after you presented your solution. You then have to adjust your solution accordingly.

good python code example:

```

"# Import necessary modules
import os

# Define the input and output file paths
input_file_path = \'numbers.txt\'
output_file_path = \'average.txt\'

# Function to read numbers from a file
def read_numbers_from_file(file_path):
    # Initialize an empty list to store the numbers
    numbers = []
    # Open the file in read mode
    with open(file_path, \'r\') as file:
        # Read each line in the file
        for line in file:
            # Strip any leading/trailing whitespace and convert to float
            number = float(line.strip())
            # Append the number to the list
            numbers.append(number)
    # Return the list of numbers
    return numbers
main()

```

Figure 2: Refining the prompt to provide comments for the generated model (slightly differently prompted).

3. a hint of the behavior of the tool ("The python REPL..."),
4. a hint of how to use the tool ("Use this..."),
5. to focus on that tool ("use only").

Prompting Semantic Verification: To verify, if a semantic correct solution can be computed with the generated constraint model, we include a verification scheme that builds on tests. Those are specific for the constraint problem at hand. However, the general prompt is enhanced with a call to the verification method, which is parameterized with the name of the constraint problem. The verification scheme is given by an example that explains the parameterization, a constraint model, and the call to the verification method (Figure 1).

Prompt Refinement: For more understanding, of why the LLM generates a certain code line in the model, we have enforced the LLM to provide comments above a code line (see Figure 2). This is done by explaining in the prompt what a "Good python code example"¹¹ is. This is done, by repeating this text at various appropriate positions in the prompt, as well as providing such an example in the prompt.

4.2. Prompting Specific Problems

The query part simply consists of the name of the constraint problem and related specific problem

¹¹The italic style in the prompt is only for the human reader, i.e., only the text is given as prompt.

Listing 1: Queries for specific constraint problems

```
queen3 = "solve the 3-queen problem"
queen8 = "solve the 8-queen problem"
queenn = "solve the n-queen problem, use n=8 as test instance"

-----

coloring = """
solve the map coloring problem for four regions, three colors and the given
adjacency:
regions = ['A', 'B', 'C', 'D']
adjacency_list = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
    'D': ['B', 'C']
}.
the solution should be a list of python dicts where each dict maps regions to
color indices
"""

-----

tsp = """solve the traveling salesman problem, use the following problem
instance:
# Number of cities
C = 4

# Distance matrix
D = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
the solution should be a list of valid solutions, each solution being a list of
integers representing the cities to be visited.
"""
```

descriptions, see Listing 1¹².

N-Queens Problem

Problem: Positioning n queens with possible movements known from chess on a $n \times n$ chessboard¹³. Figure 1 shows the used prompt.

Results: The LLM makes multiple Python REPL calls, one that prints the solution, and one that follows the system prompt on how to verify the solution. This leads to the correct computation of the model including the conversion of the solutions to a list of lists (see Figure 3).

Comments: The solution to the N-queens problem is provided in the PyChoco documentation on the internet, hence, the LLM was trained with it. However, a different solution was found. Also, the 3-queen problem which provides no solution was correctly represented. A further observation was that the result is not deterministic, e.g., it varies arithmetical constraints for diagonals (see Listing 2). Fixing the seed and setting the temperature to

zero does not make results deterministic.

The semantic verification test for the N-Queens problem is shown in Listing 3. Through the method `verify_solutions` called by the prompt, each solution (`queens`) is tested. This test is exhaustive because it analytically tests the queens' position, not leaving out a constraint.

Map Coloring

Problem: Listing 1 describes the well-known Map Coloring as a query.

Results: A first model which contains the constraint `all_different` raised an error¹⁴ but was corrected by using another constraint modeling (i.e., with `!=` instead of `all_different`) through 2 iterations¹⁵. A further run provides a one-shot success, however not recognized as such, instead a solution was hallucinated for presenting to the user¹⁶. A further run firstly used the API wrongly, but finds

¹²The code in the GitHub contains further problems such as magic square that lead to similar observations.

¹³See files "chats/queen8_cs_agent_sol2_gpt-4-1106-preview_" in the GitHub for results

¹⁴It seems to be an implementation issue, see <https://github.com/chocoteam/choco-solver/issues/705>

¹⁵coloring_cs_agent_sol2_gpt-4-1106-preview_20240529094847.txt

¹⁶coloring_cs_agent_sol2_gpt-4-1106-preview_20240529095157.txt


```

Assistant:
There was a mistake in the code. It seems I incorrectly used the
'solver.find_solution()' method, which only finds one solution, and I
didn't convert the solution to the expected list format before passing
it to the verification function. I need to find all solutions and convert
each solution to a list of integers representing the positions of the
queens.
Let me correct the code.

```

```

Python_REPL (tool call 1/1):

from pychoco.model import Model
from test_tool import verify_solutions
model = Model("8-Queens")
# Create an array of 8 variables representing the positions of queens on a
# chess board
queens = model.intvars(8, 1, 8)
# Add constraints
for i in range(8):
    for j in range(i + 1, 8):
        # Queens cannot be on the same row
        model.arithm(queens[i], "!=" , queens[j]).post()
        # Queens cannot be on the same diagonal
        model.arithm(queens[i], "-", queens[j], "!=" , i - j).post()
        model.arithm(queens[i], "-", queens[j], "!=" , j - i).post()
# Find all solutions
solver = model.get_solver()
solutions = []
for solution in solver.find_all_solutions():
    solutions.append([solution.get_int_val(queens[i]) for i in range(8)])
verify_solutions(solutions, "queen8")

```

```

Python_REPL (tool response):
92 of the 92 solutions are correct. Expected 92 solutions
Assistant:
The 8-queens problem has been successfully solved, and all 92 solutions are correct.
tool calls: 6, text messages: 6

```

Figure 3: Constraint model for 8-queens problem and last output

finally a correct solution¹⁷.

Comments: The solution can also be found on the internet, however, this does not prevent from making errors. Finally, also because the iterative approach is used, a solution was found.

Traveling Salesperson Problem

Problem: Listing 1 describes the well-known TSP as a query.

Results: In general, there are several recurring issues when attempting to model constraints for the TSP (see the error logs in Appendix Section 8). One major problem lies in accurately modeling distance constraints, which often leads to incorrect solutions. Similarly, the Hamiltonian path constraints are dif-

ficult to model correctly, compounding the problem. Additionally, there is frequent misuse of the API. These misuses typically involve providing arguments in the wrong order or supplying an incorrect number of arguments. As a result of these issues, the solutions generated are often flawed. With GPT-4, no solutions could be found with the resources at hand, but with GPT-4o, some solutions were correctly identified. However, no implementation was found that computes all the test cases provided for verification. Overall, these challenges underscore the complexity of effectively modeling TSP constraints and highlight areas that require significant improvements to find optimal solutions consistently.

¹⁷coloring_cs_agent_sol2_gpt-4-1106-preview__20240529095612.txt

Listing 2: Non-deterministic results

```
- correct
    model.arithm(queens[i], "!=" , queens[j] + (j - i)).post()
    model.arithm(queens[i], "!=" , queens[j] - (j - i)).post()
- correct
    model.arithm(qs[i], "-", qs[j], "!=" , j - i).post()
    model.arithm(qs[j], "-", qs[i], "!=" , j - i).post()
- wrong
    model.abs(queens - queens).ne(i - j).post() # Diagonal constraint
    model.abs(queens - queens).ne(j - i).post() # Diagonal constraint
- wrong
    model.arithm(queens[i], "-", queens[j], "!=" , i - j).post()
    model.arithm(queens[i], "+", queens[j], "!=" , i + j + 2).post()
- does not find all solutions
    diag1 = [model.int_offset_view(queens[i], i) for i in range(n)]
    diag2 = [model.int_offset_view(queens[i], -i) for i in range(n)]
    model.all_different(diag1).post()
    model.all_different(diag2).post()
- wrong
    model.arithm(queens[i], "!=" , queens[j]).post() # Different columns
    model.arithm(queens[i], "-", i, "!=" , queens[j], "-", j).post() #
        Different diagonals (left-top to right-bottom)
    model.arithm(queens[i], "+", i, "!=" , queens[j], "+", j).post() #
        Different diagonals (left-bottom to right-top)
- wrong
    model.all_different([queens[i], queens[j]]).post()
    model.arithm(queens[i], "-", queens[j], "!=" , j - i).post()
    model.arithm(queens[i], "+", queens[j], "!=" , j - i).post()
```

Listing 3: Verifying a solutions for N-queens through testing the queens' positions.

```
def is_valid_solution(queens):
    n = len(queens)
    assert type(queens) == list, "expected list of ints"
    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are in the same row
            if queens[i] == queens[j]:
                return False
            # Check diagonals: if the difference between the column indices
            # equals
            # the difference between the row indices, they're in the same
            # diagonal
            if abs(queens[i] - queens[j]) == abs(i - j):
                return False
    return True
```

4.3. Implementation

The implementation, which utilizes LangChain and Python leverages ChatGPT-4 and ChatGPT-4o and is available on GitHub¹⁸. As constraint solver, we used PyChoco as a Python API for the constraint solver Choco¹⁹. The resources of ChatGPT are limited in respect of context/prompt length. This

leads to simply stopping answering by the LLM if the chat, which is with each answer extending, i.e., e.g., in the last prompt the previous chat history is part of the prompt. The resulting chats are added to the paper as an appendix as well as in the GitHub. The implementation can be used to rerun the experiments, and the data can be utilized for further investigation in prompt engineering.

¹⁸<https://github.com/hitecev/iCMLLM>

¹⁹<https://choco-solver.org/>

5. Discussion

Overall, across all problems, after initiating multiple chats, each with several iterations, a model that can compute a single solution could be found (see Appendix sections 3 to 6). In Appendix Section 9, we document the number of iterations per experiment, i.e., calls to the Python interpreter ("Tool Calls") and answers given by the LLM ("Text Messages"). Nevertheless, for the TSP, we could not find a model that computes all solutions. The iteration was stopped by the LLM, if no changes occurred or if too many tokens were used. So the paper's result is that in principle constraint models for well-known problems can be computed, however, not in any cases.

An interesting observation is that the generated variable names within the model are always semantically meaningful (e.g., "queens") and not arbitrary. This likely occurs because LLMs statistically favor names that have been previously encountered.

In this paper, we use well-known constraint problems, which, of course, can only be a first step for leveraging LLMs in knowledge acquisition. The next steps would be to supply new, unknown problems, also industrial ones, e.g., by giving tables of correct variable combinations (configurations) in table constraints and generating abstracted constraint models. However, Listing 1 shows a potential way of representing formally a problem as a query which probably can be used for other tasks.

There are several general problems identified: the program persistently uses the PyChoco API incorrectly and often hallucinates PyChoco functions. This could not be prevented by providing the API definition in the prompt, because the complete definition extends the number of possible tokens we could send to the LLM.

Additionally, the chats are not stable and can vary significantly even with the same prompt. This instability persists even with temperature set to 0 and a fixed seed, making chats non-deterministic.

The used prompts should be further developed to achieve the intended general generation of knowledge models; in other words, prompt engineering for knowledge modeling needs to be elaborated. Few-Shot Prompting [20] could be used to encourage the model to produce more consistent outputs and adhere more closely to given instructions.

The approach of first creating syntactically correct models and then verifying them to ensure semantic accuracy could be successfully demonstrated (see Listing 3). Because we use a test-driven semantic verification this verification depends on the quality of the used test cases. For example, the N-queen

problem was exhaustive because of an analytical-based verification and not an enumeration of test cases. Furthermore, writing tests for knowledge-based tasks is, in our experience, typically easier than writing knowledge models, hence, the experts' tasks of creating formal constraints and also configuration models will, in our opinion, shift from writing models to writing tests - and let us write the models through an LLM. Thus, the knowledge model developing process probably, with the presented approach, can be executed by domain experts not only by knowledge engineers.

The paper offers, furthermore, an easy way to begin with iterative LLM prompting. Moreover, it is suggested to use the LLM for creating a new configuration language by exploiting the hallucination capabilities.

6. Conclusions

The paper presents the first steps in supporting the formalization task in constraint modeling with LLMs, i.e., the mapping of a well-known problem into a constraint representation of a certain constraint language, here PyChoco, with large language models. By taking various constraint problems, formulating generally applicable prompts, and doing so in an iterative manner which includes syntactic and semantic verification processes, the LLM could generate appropriate and correct constraint models. Through the iteration, in-between errors related to syntax and semantics were automatically corrected. Further work will include a quantitative evaluation of the method, a comparison of different LLMs, as well as the generation of constraint models for unknown problems. In total, this paper provides a first step into a new kind of constraint and configuration modeling approach with LLMs.

Acknowledgments

This project was funded through the Federal Institute for Research on Building, Urban Affairs and Spatial Development (Bundesinstitut für Bau-, Stadt- und Raumforschung), Germany, within the Zukunft Bau Forschungsförderung Maßnahme (Future of construction Research funding Measure): "AI-supported modernization of rental housing stock" ("KI-gestützte Modernisierung an Mietwohnungsbaubeständen (KIMM)") as well as through the Federal Ministry for Economic Affairs and Climate Action (Bundesministerium für Wirtschaft und Klimaschutz), Germany, within the "Joint project: EnOB: IntelMOD - Intelligent modern-

ization platform based on functional cost splitting; sub-project: Infrastructure of the modernization platform.“ (“Verbundvorhaben: EnOB: IntelMOD - Intelligente Modernisierungsplattform auf Basis des Funktionalen Kostensplittings; Teilprojekt: Infrastruktur der Modernisierungsplattform.“)

References

- [1] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen, Knowledge-based Configuration – From Research to Business Cases, Morgan Kaufmann, 2014.
- [2] A. Felfernig, A. Falkner, D. Benavides, Feature Models: AI-driven Design, Analysis, and Applications, Springer, 2024.
- [3] L. Hotz, A. Felfernig, M. Stumptner, A. Ryabokon, C. Bagley, K. Wolter, Configuration Knowledge Representation & Reasoning, in: A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen (Eds.), Knowledge-based Configuration – From Research to Business Cases, Morgan Kaufmann Publishers, 2014, pp. 59–96.
- [4] I. Ozkaya, A. Carleton, J. Robert, D. Schmidt, Application of large language models (llms) in software engineering: Overblown hype or disruptive change?, Carnegie Mellon University, Software Engineering Institute’s Insights (blog), 2023. URL: <https://doi.org/10.58012/6n1p-pw64>, accessed: 2024-May-16.
- [5] B. O’Sullivan, Automated modelling and solving in constraint programming, in: 24th AAAI Conference on Artificial Intelligence, AAAI’10, AAAI Press, 2010, pp. 1493–1497.
- [6] E. C. Freuder, Conversational modeling for constraint satisfaction, AAAI Conference on Artificial Intelligence 38 (2024) 22592–22597. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/30268>. doi:10.1609/aaai.v38i20.30268.
- [7] F. Petroni, T. Rocktäschel, P. P. Lewis, A. Bakhtin, Y. Wu, A. Miller, S. Riedel, Language models as knowledge bases?, 2019. URL: <https://arxiv.org/pdf/1909.01066>.
- [8] Y. Ding, A. Poudel, Q. Zeng, T. Weninger, B. Veeramani, S. Bhattacharya, Entgpt: Linking generative large language models with knowledge bases, 2024. URL: <https://arxiv.org/abs/2402.06738>.
- [9] S. Razniewski, A. Yates, N. Kassner, G. Weikum, Language models as or for knowledge bases, 2021. URL: <https://arxiv.org/abs/2110.04888>.
- [10] F. Suchanek, A. T. Luu, Knowledge bases and language models: Complementing forces, in: 7th International Joint Conference on Rules and Reasoning, Springer-Verlag, Berlin, Heidelberg, 2023, pp. 3–15. URL: https://doi.org/10.1007/978-3-031-45072-3_1.
- [11] A. Nayak, H. P. Timmapathini, Llm2kb: Constructing knowledge bases using instruction tuned context aware large language models, 2023. URL: <https://arxiv.org/abs/2308.13207>.
- [12] T. Ahmed, S. Choudhury, Lm4opt: Unveiling the potential of large language models in formulating mathematical optimization problems, 2024. URL: <https://arxiv.org/abs/2403.01342>.
- [13] Y. Zhang, S. Mao, T. Ge, X. Wang, A. de Wynter, LLM as a mastermind: A survey of strategic reasoning with large language models, 2023. URL: <https://arxiv.org/abs/2404.01230v1>. arXiv:2404.01230.
- [14] W. Chen, X. Ma, X. Wang, W. W. Cohen, Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, Transactions on Machine Learning Research (2023). URL: <https://openreview.net/forum?id=YfZ4ZPt8zd>.
- [15] L. Pan, A. Albalak, X. Wang, W. Y. Wang, Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning, 2023. arXiv:2305.12295.
- [16] P. Clark, O. Tafjord, K. Richardson, Transformers as soft reasoners over language, in: 29th International Joint Conference on Artificial Intelligence, IJCAI’20, 2021, pp. 3882–3890.
- [17] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. H. Hsin Chi, D. Zhou, Self-consistency improves chain of thought reasoning in language models, ArXiv (2022). URL: <https://arxiv.org/abs/2203.11171>.
- [18] M. Acher, J. G. Duarte, J.-M. Jézéquel, On programming variability with large language model-based assistant, in: Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A, SPLC ’23, Association for Computing Machinery, New York, NY, USA, 2023, p. 8–14. URL: <https://doi.org/10.1145/3579027.3608972>.
- [19] M. Funk, S. Hosemann, J. C. Jung, C. Lutz, Towards ontology construction with language models, 2023. URL: <https://arxiv.org/abs/2309.09898>.
- [20] 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 33 (2020) 1877–1901.