Configuration Copilot: Towards Integrating Large Language Models and Constraints

Philipp Kogler[1,*], Wei Chen[1], Andreas Falkner[1], Alois Haselböck[1] and Stefan Wallner[1]

[1]*Siemens AG Österreich, Siemensstraße 90, 1210 Wien, Austria*

Abstract

A product configurator enables the configuration of a customizable product while constraining possible variations. Users typically interact with a product configurator via a graphical user interface. A complex product can be composed of components and parameters that are not easily understandable for non-experts which can prevent them from effectively configuring the product. In this paper, we propose a configuration copilot, an interactive chat-based interface that allows users to iteratively configure a product by describing their requirements in natural language. Our framework leverages the Natural Language Processing (NLP) capabilities of advanced pre-trained Large Language Models (LLMs) alongside the robustness of constraint-based product configurators. We introduce a technical architecture that accurately formalizes constraints from natural language inputs, identifies valid product configurations based on a defined product line and specified constraints using a constraint solver, and communicates the resulting product configurations back to the end user in natural language. We demonstrate and evaluate the configuration copilot on two use-cases: The configuration of the GoPhone feature model (Boolean feature assignments), and the configuration of a metro wagon (more general configuration parameters).

Keywords

Product Configuration, Constraints, Feature Models, Large Language Models, Copilot

1. Introduction

Product configuration involves creating customized products from predefined components while satisfying constraints that limit configurable parameters and possible combinations [1]. A product configurator is a software tool that allows users to configure a product, commonly through a graphical user interface and often in a web-based context. Therefore, interface and interaction design plays a major role in the development of a product configurator but is often overlooked [2]. This observation is especially relevant when complex products are configured by non-expert users. The meaning of configurable components and parameters may not be obvious which prompts a need for explanation and introduces a learning curve.

As an alternative to GUI-based interactions with product configurators, we propose a configuration copilot that offers a text-based chat interface. Uninformed users shall be able to describe their requirements in natural language without knowledge of the concrete parameters to set and components to select. The copilot shall then configure the product and respond with a valid configuration complying with the initial requirements. The user shall be able to interactively refine the product configuration.

We utilize a pre-trained Large Language Model (LLM) for the processing of natural language. Recent advances in this field have enabled use cases that require the understanding and generation of not only natural language but also code. Well-known limitations include a lack of reliability, guaranteed correctness, domain-specific knowledge in general-purpose LLMs, and limited reasoning abilities [3]. In our configuration copilot, we address these shortcomings by combining a LLM with a constraint solver. While the strengths of the LLM are utilized in the processing of the natural-language requirement descriptions, the reasoning to find valid configurations is done by the constraint solver.

In this paper, we first describe LLMs and constraint-based product configuration in Section 2 and related work in Section 3. We detail the technical architecture of the configuration copilot in Section 4, and present an evaluation based on the two use-cases of configuring the GoPhone feature model and a metro wagon in Section 5. We conclude the paper with a summary, a limitation statement, and future work in Section 6.

2. Background

2.1. Large Language Models

Pre-training task-agnostic aspects of natural language processing (NLP) tasks is a central concept of LLMs. The *Transformer* architecture enables this approach on a large scale through parallelization. Transformer

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

*Corresponding author.

philipp.kogler@siemens.com (P. Kogler); chen.wei@siemens.com (W. Chen); andreas.a.falkner@siemens.com (A. Falkner); alois.haselboeck@siemens.com (A. Haselböck); stefan.wallner@siemens.com (S. Wallner)
0009-0009-5598-1225 (P. Kogler); 0009-0008-0486-9068 (W. Chen); 0000-0002-2894-3284 (A. Falkner); 0000-0003-2599-3902 (A. Haselböck); 0000-0002-9755-6632 (S. Wallner)

models are able to capture complex patterns and long-range-dependencies in texts through the multi-head self-attention mechanism. Compared to previous state-of-the-art models such as recurrent neural networks (RNNs) or long short-term memory networks (LSTMs) a performance improvement in various NLP tasks is observed [4, 5].

Decoder-only models are a subclass of Transformer-based architectures and are primarily used for sequence-to-sequence tasks such as translation. Auto-regressive models predict the next single token (sub-word) by maximizing the log-likelihood given all previous words and the model parameters [4].

The size and quality of the pre-training corpus have a strong impact on performance [5]. LLMs are trained on publicly available data and excel in general language tasks. Highly specialized tasks require expert knowledge that is often not included in the training data, and therefore LLMs may not be able to generate accurate output. Task-specific knowledge can be introduced to a general-purpose LLM through domain customization by employing techniques like *prompting* and *fine-tuning* [6].

2.2. Constraint-based Product Configuration

Product configuration involves selecting and assembling various components and options to meet customer requirements and constraints. Its complexity arises from the vast number of possible combinations and the need to satisfy all technical restrictions and customer preferences. To handle this complexity, powerful technologies have been developed and established in the last decades. Constraint-based systems shall be highlighted here, which allow to represent the product line and its technical restrictions and requirements in a clean, logical way, thereby ensuring that only valid configurations are generated. The core of such systems lies in the ability to handle complex and combinatorial search spaces efficiently through the use of advanced solving algorithms, such as backtracking, forward checking, and constraint propagation. This facilitates the efficient generation of feasible solutions while pruning invalid combinations.

An important subdomain of configuration problems are feature models for the representation of product lines [7]. Constraint-based techniques are especially well-suited for such feature models, because of the simple language and the mainly Boolean type of the variables.

MiniZinc is a constraint language that can be used to represent configuration problems [8]. Several efficient solvers can process this language and can therefore be used as the backend of a configurator.

A product configurator is almost always an interactive system [9]. A graphical user-interface (GUI) allows to enter the user requirements, which are passed on as input to the constraint solver. The results of the solver are presented on the GUI, and the user can vary or refine her/his input specification and the solver is called again.

To design and implement a configurator GUI can be a challenging task, because the possible interactions are diverse, like collecting the requirements, reporting invalid constellations, representing a solution, showing a performance value of a solution, etc. In addition, every modification of the product (line) requires a review and possibly and adjustment of the GUI.

In the following sections, we demonstrate how to eliminate the need for a product-specific GUI by utilizing an LLM to engage in dialogue with the user.

3. Related Work

Various approaches to improve the reliability, the performance in domain-specific tasks, and the reasoning abilities of LLMs are described in literature.

Few-shot prompting effectively introduces domain-specific knowledge and improves the task-specific performance of LLMs by adding a small set of example interactions (input and expected output) to the prompt [10]. *Chain-of-thought prompting* was shown to improve the reasoning abilities of LLMs especially in more complex tasks by providing exemplary intermediate reasoning steps [11].

Grammar prompting is used when a specific output format is expected. Wang et al. describe how a minimal specialized grammar is obtained in a grammar specialization process by selecting a specialized grammar as a subset of the full grammar using an LLM and minimizing it by parsing the output and forming the union of used rules. In their approach, constrained decoding then validates the output syntax [12]. Similarly, Poesia et al. presented the *Synchromesh* framework: Using a few-shot prompting technique, semantically similar examples are selected from a larger pool for a given natural language prompt via a similarity metric named *Target Similarity Tuning*. Constraints are enforced through *Constrained Semantic Decoding* to verify syntax validity, scoping, or type checks. During the token-by-token construction of the LLM output, a *Completion Engine* provides all valid tokens that can further extend a partial program towards a full correct program [13].

Neuro-symbolic approaches focus on combining the strengths of neural networks and symbolic reasoners. Pan et al. introduced the *Logic-LM* framework which achieves a performance improvement of 18% on logical reasoning datasets over chain-of-though prompting. The framework translates the natural-language input into symbolic formulations and utilizes a symbolic reasoner to obtain the answer [14].

This paper builds upon our previous work [15] that studied the reliable generation of formal specifications with LLMs using algorithmic post-processing. We extend the approach towards product configuration by applying post-processing to reliably integrate a constraint solver. In addition to previously described guaranteed syntactically valid output, this extension enables arbitrary semantic constraints.

4. Configuration Copilot

This section presents the technical details of the configuration copilot that combines LLMs with constraint-based configuration.

4.1. Architecture

Figure 1 shows an overview of the architecture. A user configures a product by providing a natural-language description of their requirements. The Formalizer (see Section 4.2) is a specialized LLM-based component that translates the requirements to constraints. The Configuration Engine is a constraint solver that attempts to find a configuration that satisfies the general constraints of the product line combined with the user constraints provided by the Formalizer. An Interpreter (see Section 4.4) translates the configuration back to natural language. The configuration copilot then responds with a natural-language description of the configured product accompanied by the full technical specification (product configuration as determined by the Configuration Engine). The user can then further refine the product configuration interactively.

4.2. Formalizer

The input to the Formalizer is a natural-language description of arbitrary product requirements provided by a non-expert. Utilizing the NLP capabilities of LLMs, the formalization can be viewed as a sequence-to-sequence translation task from natural language to a formal specification. The LLM is tasked with natural language understanding and the identification of corresponding parameters or components of the product (line), but is specifically not tasked with reasoning (e.g., constraint satisfaction). While pre-trained LLMs achieve a strong performance on general tasks, they do not have knowledge of the specific product (line) to configure as corresponding data is not included in their training corpus [5, 6]. Additionally, the probabilistic nature of the token-by-token output construction of LLMs does not provide any guarantees in the correct generation of valid constraints [4].

Our framework for reliable code generation addresses domain-customization and reliable output generation through few-shot prompting and algorithmic post-processing [15].

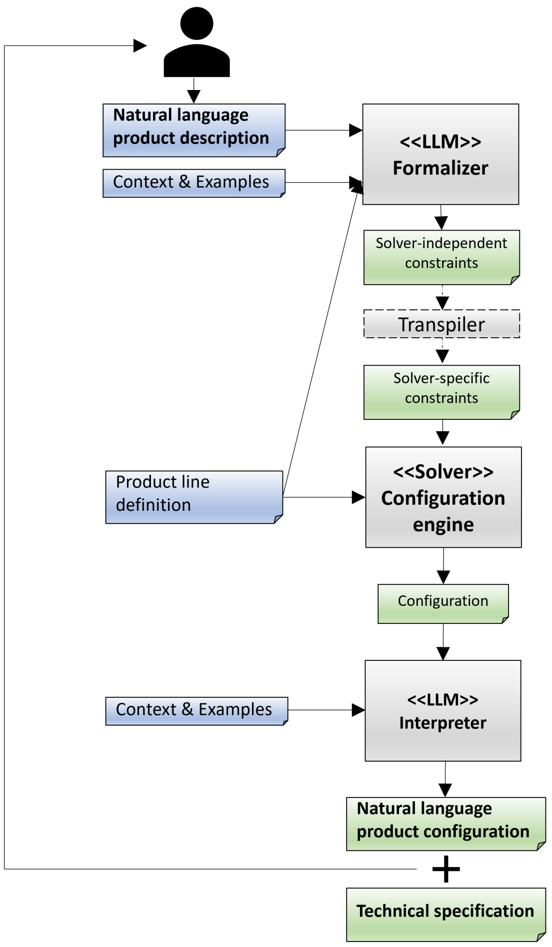
Natural language product description
Context & Examples
<<LLM>> Formalizer
Solver-independent constraints
Transpiler
Solver-specific constraints
Product line definition
<<Solver>> Configuration engine
Configuration
Context & Examples
<<LLM>> Interpreter
Natural language product configuration
+
Technical specification

Figure 1: Architecture of the Configuration Copilot

Few-shot prompting has been shown to effectively extend the capabilities of LLMs with domain knowledge while requiring significantly less training data than fine-tuning [10]. Knowledge of the product line is incorporated through a system prompt describing the product line with its parameters and components. A small set of examples is appended as pairs of natural-language inputs and expected outputs to provide the LLM with more context and guide it towards the expected behavior.

Rather than generating output directly in a specific constraint language, an intermediary JSON-based language is used, which can then be easily transpiled. The transpiler parses the JSON constraint representation and maps its elements to corresponding constructs of the specific constraint language following predefined rules. As JSON is widely used, pre-trained LLMs have more often encountered JSON than less common constraint languages. Therefore, the generation of an intermediary JSON output is closer to the LLMs capabilities. Additionally, an intermediary language gives more control over the expected output as available language constructs can be constrained and tailored to the specific task. It also decouples the Formalizer from the Configuration Engine by enabling interchangeability of the concrete constraint language. To generate a valid JSON for the Formalizer, several state-of-the-art LLMs are evaluated and benchmarked. Specialized code LLMs that are pre-trained on the translation of natural language to code in a variety of programming languages are believed to be more suitable for the generation of structured JSON output. In our evaluation in Section 5, we selected four open-access LLMs: Two code LLMs (CodeLLama [16] and Codestral [17]), and two general-purpose instruction-tuned LLMs (Meta Llama 3 [18] and Mistral [19]).

Algorithmic post-processing guarantees the correct generation of the JSON-based intermediary language and is depicted in Figure 2. As the auto-regressive Transformer model generates its output step-by-step as tokens, the post-processor engages into every generation step: For each step, the LLM generates a list of candidates for the next token based on the prompt and the generated output so far. Sorted by priority as evaluated by the LLM, the post-processor determines whether the token candidate represents a valid continuation of the partial output sequence (partial intermediary JSON). The valid token candidate with the highest priority is then selected, handed back to the LLM, and added to the partial JSON, extending it one step further. A completeness checker evaluates after every step to determine, if the JSON is complete [15].

The JSON-based intermediary language is formally defined by a JSON schema specification and the post-processor is therefore a specialized JSON validator that can strictly validate any partial JSON against the schema. This implementation is based on deterministic finite automata (DFA). Each generic JSON language element (object, list, string, number, etc.) is represented by a DFA, keeping track of the current state. The token generated by the LLM is broken down to single-character inputs for the JSON validator. Depending on the schema and the current state, only a set of characters is accepted. If a character is rejected, the current token is considered invalid, and the validator state is rolled back to the last valid token. State changes are triggered by characters until the final state is reached. When the DFA reaches its final state, the generated valid JSON is complete [15].

4.3. Configuration Engine

Given the user constraints combined with the complete product line definition, the Configuration Engine evaluates whether the constraints are satisfiable and returns a configuration. The product line as well as the user constraints are modelled in the MiniZinc constraint language [8]. The solver returns the full product configuration as a list of variable assignments which serves as an input to the Interpreter. In this context, we consider the constraint solver a given technology that will neither be further described nor evaluated.

4.4. Interpreter

The Interpreter is an LLM module that explains the product configuration found by the Configuration Engine. The goal is to provide the user with a less technical summary that is understandable for non-experts.

Structured few-shot prompting [10] is sufficient for this use-case as LLMs generally perform well in the translation from a formal specification to a natural-language summary as all facts are directly present in the prompt. The context given to the LLM consists of three aspects: The product line definition, instructions, and examples. The LLM is prompted to evaluate which properties and components are most important to be included in the summary. This is achieved by adding importance hints to the product line definition, and by appending the original user input. Properties and components mentioned directly in the user input are given more importance and are more likely to be included in the summary. The result is a more natural context-aware explanation of the most relevant aspects in the product configuration.

5. Evaluation

The presented configuration copilot is evaluated on two use-cases: The conceptually simpler task of configuring a feature model, and the configuration of a metro Wagon.

5.1. Feature Model (GoPhone)

The first use-case for the evaluation of the presented copilot is the configuration of a feature model. An uninformed user shall be supported in the configuration of the GoPhone from the SPLOT project [20].

The GoPhone is a feature model comprised of 77 features with some being mandatory, optional, dependent on other features, or mutually exclusive. For example, the feature `call` is mandatory for the GoPhone, the feature `accept_incoming_call` is mandatory for `call`, but `show_missed_calls` and `show_received_calls` are optional.

Feature assignments are Boolean, either the feature is included in the product configuration (`true`) or the feature is not included (`false`). The product line definition is a MiniZinc program that was directly derived from the

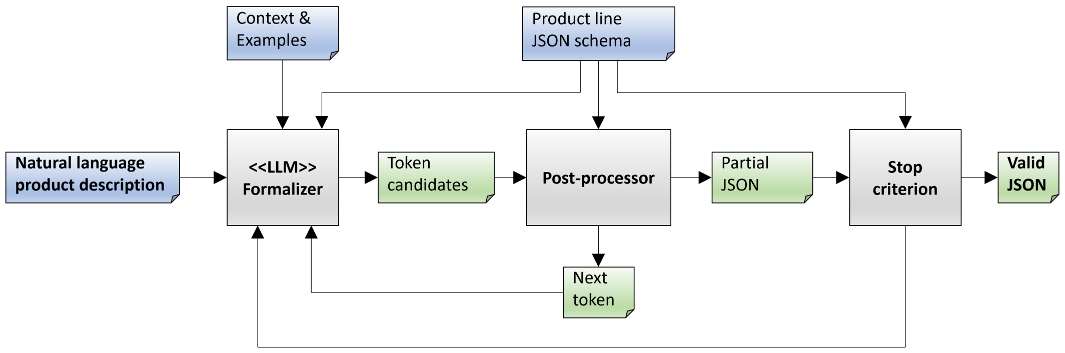

Figure 2: Detail view of the Formalizer with post-processing

feature model. Each feature is a Boolean variable. Constraints limit the combination of features and therefore limit possible product configurations.

A non-expert user starts by describing their requirements for the phone in natural language:

```
I need a basic phone to call people and browse
the web but I don't play games. I also want to
keep track of my appointments.
```

This natural-language description is then formalized to the intermediary JSON language:

```
{
    "features": [
        {
            "name": "make_call",
            "value": true
        },
        {
            "name": "browsing",
            "value": true
        },
        {
            "name": "game",
            "value": false
        },
        {
            "name": "calendar_entry",
            "value": true
        }
    ]
}
```

This list of solver-independent constraints is then transpiled to MiniZinc constraints:

```
constraint make_call = true;
constraint browsing = true;
constraint game = false;
constraint calendar_entry = true;
```

Together with the MiniZinc program (product line definition), the Configuration Engine evaluates the constraints and returns a full product configuration of the GoPhone for the specific user requirements as a list of Boolean feature assignments. In this work, the Gecode [21] solver was used without further configuration or optimization. The Interpreter converts this configuration back to natural language and returns it to the user. An example for such an output is (the technical specification is shortened for brevity):

```
Your GoPhone can manage ringing tones, messages,
and browse the web. It can also manage calls,
read multimedia, and display photos. It has a
calendar entry feature and an address book
processing system. However, it does not play
games, organize tasks, or have currency
conversion features.

Here is the full technical configuration:

GoPhone = true;
manage_ringing_tones = true;
[...]
browse = true;
[...]
game = false;
play_games = false;
install_games = false;
[...]
```

The crucial and potentially failing component of the presented architecture is the Formalizer: the probabilistic nature of the underlying LLMs does not provide strict guarantees. Especially the translation of the user's requirements to the feature assignments is subject to uncertainty. A formal evaluation of the Interpreter is not done because the correctness requirements for the configuration summary are less strong and LLMs are generally known to perform well on simple summarization tasks when the facts are directly provided. It is also unsuitable to define a single reference solution as a large variety of summaries (with various feature assignments being explained or not explained) could be considered correct. Ultimately, users needs to decide whether the summary was helpful or not.

The Formalizer was evaluated on a custom dataset of 30 test cases. 15 test cases create a new configuration from scratch, and 15 test cases evaluate a re-configuration where a given configuration is modified. Each test case consists of natural-language input mentioning between two and six feature requirements in the text (and up to 30 given feature assignments for modification test cases), and the expected feature assignments in JSON. Using the natural-language input, the Formalizer generates feature assignments in JSON. This output is compared to the expected output. The comparison is conceptually challenging due to the intrinsic ambiguity of natural language. In many cases, one could argue for multiple options of feature assignments to be considered a correct translation. In this evaluation, we hand-crafted the dataset to be less ambiguous. However, the features of the GoPhone are in themselves sometimes not obviously distinguishable, and multiple features may be equally suitable. For example, the feature `browsing` is an optional sub-feature of the more general parent feature `browse`. This ambiguity was addressed by encoding very similar features to the same representation. Therefore, all defined synonymous features are considered a correct feature assignment for a requirement. However, the feature assignment was not limited to leaf features because doing so would add reasoning requirements to the Formalizer. Consider the leaf features `play_games` and `install_games`, and the parent feature `game`. If a user only mentions games in their descriptions, the more abstract feature `game` shall be assigned. Otherwise, the LLM would have to reason about a proper assignment of leaf features, deviating from the most direct translation from natural language to a feature assignment. The reasoning regarding further (sub-)feature assignments shall be done by the Configuration Engine.

A similarity metric based on the Jaccard distance between sets [22] was used to compare each pair of expected and actual output: Let T and F be the sets of feature names in the expected output where the feature value is *True* and *False*, respectively. Similarly, let $\hat{T}$ and $\hat{F}$ be the sets of feature names in the actual output where the feature value is *True* and *False*, respectively. The Jaccard similarities are:

For the *True* sets:

$$S_T = \frac{|T \cap \hat{T}|}{|T \cup \hat{T}|}$$

For the *False* sets:

$$S_F = \frac{|F \cap \hat{F}|}{|F \cup \hat{F}|}$$

Table 1
Evaluation Results for the GoPhone Formalization
S = Similarity score
F1 = F1 score

Model [Size/Quantization]	S	F1
CodeLlama 34B/Q4 [Link]	0.65	0.74
Codestral 22B/Q4 [Link]	**0.79**	**0.86**
Meta Llama 3 8B/Q8 [Link]	0.46	0.58
Mistral 7B/Q8 [Link]	0.69	0.79

The overall similarity between the expected and actual output S as the weighted average of S_T and S_F is:

$$S = \frac{S_T \cdot |T \cup \hat{T}| + S_F \cdot |F \cup \hat{F}|}{|T \cup \hat{T}| + |F \cup \hat{F}|}$$

The result is a number between 0 and 1, with 0 indicating no similarity, and 1 indicating a perfect match. In this metric, the identification of features in the natural language as well as the Boolean assignment are considered.

Similarly, the precision P, recall R and F1 score $F1$ were calculated:

$$P = \frac{|T \cap \hat{T}| + |F \cap \hat{F}|}{|\hat{T}| + |\hat{F}|}$$

$$R = \frac{|T \cap \hat{T}| + |F \cap \hat{F}|}{|T| + |F|}$$

$$F1 = 2 \cdot \frac{P \cdot R}{P + R}$$

We selected four open-access LLMs from HuggingFace to be evaluated in the context of the configuration copilot, two code models, and two general-purpose models. Table 1 summarizes the evaluation results for the GoPhone use-case per LLM. Codestral 22B/Q4, a state-of-the-art code model, performed best. However, Mistral 7B/Q8 outperformed the larger code model CodeLLama 34B/Q4 against our expectations. This shows that the performance of LLMs is use-case specific and must be evaluated. We found that the performance degrades as instances become more complex. Remedies for this observation are the use of larger models, tuning the technical approach, or future improvements of LLMs themselves. Considering the remaining ambiguity of natural language, the results indicate reasonable performance in this use-case as the majority of feature requirements was formalized correctly.

5.2. Metro Wagon

The second use-case for the evaluation is a metro Wagon configuration problem (see [23]) that uses not only Boolean but also numeric variables and arrays, where a configurable product has components that can occur

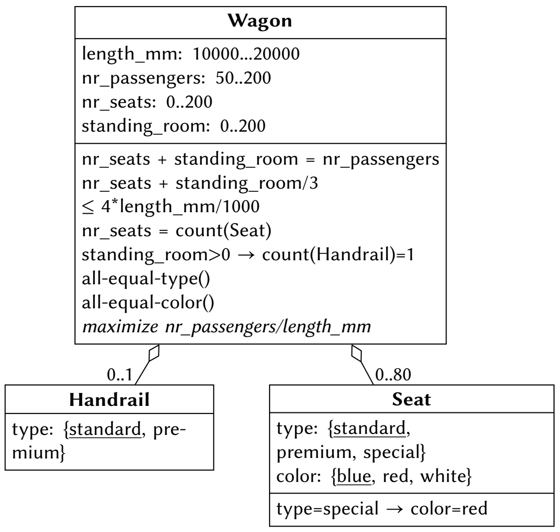
Wagon
length_mm: 10000...20000
nr_passengers: 50..200
nr_seats: 0..200
standing_room: 0..200
nr_seats + standing_room = nr_passengers
nr_seats + standing_room/3
≤ 4*length_mm/1000
nr_seats = count(Seat)
standing_room>0 → count(Handrail)=1
all-equal-type()
all-equal-color()
maximize nr_passengers/length_mm
0..1
0..80
Handrail
type: {standard, premium}
Seat
type: {standard, premium, special}
color: {blue, red, white}
type=special → color=red

Figure 3: Class diagram of the Wagon example. *Default values are underlined. Wagon.all-equal-type() stands for a constraint that all sub-parts must have the same type except for special. Wagon.all-equal-color() stands for a constraint that all associated seats (except if type=special) must have the same color.*

multiple times (similar to generative constraint satisfaction [24] or cardinality-based feature modelling [25]).

A metro train wagon has as configurable attributes the size (length in millimetres: 10000..20000) and the expected load (number of passengers: 50..200) which can be realized as seats or standing room. As components we consider only seats (max. 4 per meter of length) and handrails, and their number is configurable.

There is at most one handrail in a wagon (mandatory if there is standing room) and it has a configurable type: "standard" or "premium".

A single seat consumes standing room for 3 persons and has as configurable attributes the type ("standard", "premium", "special") and the color ("blue", "red", "white"). The type is constrained such that standard is not allowed to be mixed with premium (for seats and handrails). The color of all seats must be the same, except for special seats which have to be "red".

Figure 3 shows a UML class diagram for this sample specification, including pseudo code for all constraints.

A non-expert user starts by describing their requirements for the metro Wagon in natural language:

```
The wagon should accommodate more than 120 people
 with room for 40 to sit. Seats should be red.
```

This natural-language description is then formalized to the intermediary JSON language:

```
{
   "nr_passengers": {
       "type": "greaterThan",
       "value": 120
   },
   "nr_seats": {
       "type": "equals",
       "value": 40
   },
   "seat_color": [
       "red",
       "red",
       "red",
       ...
   ]
}
```

This list of solver-independent constraints is then transpiled to MiniZinc constraints:

```
constraint nr_passengers > 120;
constraint nr_seats = 40;
constraint forall (i in 1..nr_seats)
   (seat_color[i] = red);
```

Together with the MiniZinc program (product line definition), the Configuration Engine evaluates the constraints and returns a full product configuration of the metro Wagon for the specific user requirements as a list of value assignments to the configurable parameters. The Interpreter converts this configuration back to natural language and returns it to the user:

```
Your metro Wagon is 20 meters long, has space for
 160 passengers with 40 red standard seats and a
standard handrail. There is also standing room
for an additional 120 people.

Here is the full technical configuration:

length_mm = 20000;
nr_passengers = 160;
nr_seats = 40;
standing_room = 120;
nr_handrails = 1;
handrail_type = standard;
seat_color = [red, red, red, ...];
seat_type = [standard, standard, standard, ...];
```

The Formalizer for the metro Wagon use-case was evaluated, like the GoPhone Formalizer, on a diverse set of 30 test cases (pairs of input and expected output) with 15 creating a new configuration and 15 modifying a given configuration (re-configuration). To evaluate the similarity in this use-case, the previously described similarity metric based on the Jaccard distance between sets for Boolean feature assignments was extended to the more general use-case. This extension is necessary to enable the evaluation of the value assignment for the

Table 2
Evaluation Results for the Metro Wagon Formalization
S = Similarity score

Model [Size/Quantization]	S
CodeLlama 34B/Q4 [Link]	0.77
Codestral 22B/Q4 [Link]	**0.78**
Meta Llama 3 8B/Q8 [Link]	0.68
Mistral 7B/Q8 [Link]	0.72

extended variable types (i.e., strings, numbers, arrays). While the Jaccard distance remained the basis for the similarity metric, a type-specific value metric was applied to each configuration parameter that is present in both, the expected and the actual output. In addition to the parameters being present, the total similarity is adjusted according to the value similarities as well. The type-specific metric considers:

- for numeric values: operator ('=', '>', '<', etc.) and value distance relative to the parameter-specific domain (value range)
- for array values: length and positional item equality
- for string-enumerated values: exact value match

Let C be the set of configuration parameter names in the expected output and let $\hat{C}$ be the set of configuration parameter names in the actual output. The Jaccard similarity S_J is:

$$S_J = \frac{|C \cap \hat{C}|}{|C \cup \hat{C}|}$$

Let c be a matching parameter that is in both, the expected and the actual output, and let $S_v(c)$ be the type-specific value similarity (between 0 and 1) of c between the expected and the actual output. The value-adjusted Jaccard similarity S is then:

$$S = \frac{\sum_{c \in C \cap \hat{C}} S_v(c)}{|C \cup \hat{C}|}$$

The evaluation of the F1 score is omitted because it does not provide any additional value, as it appears to correlate strongly with the already rather strict similarity score S. Table 2 summarizes the evaluation results for the metro Wagon use-case per LLM. Codestral 22B/Q4 performed best again with a similar score. However, the other three models consistently improved their score compared to the GoPhone use-case. While the metro use-case in itself is more complex, the domain size (amount of named parameters) is lower, which may be the reason for the higher performance. Overall, the results again indicate a reasonable performance for the metro Wagon use-case.

6. Conclusion

This paper presented a configuration copilot that enables non-expert users to configure a product in natural language. The cooperative neuro-symbolic approach combines an LLM with a constraint solver to reliably support a product configuration. An early evaluation on the two use-cases of configuring the GoPhone feature model and a metro Wagon indicated practical feasibility. We believe that a configuration copilot is a valuable extension to GUI-based product configurators. For a productive implementation, limitations and future work mentioned in Sections 6.1 and 6.2 should be addressed.

6.1. Limitations

A limitation of our work is the size of the use-cases. Compared to real-world scenarios, the evaluated GoPhone feature model and metro Wagon are smaller and less complex. Additionally, the evaluation was done on a limited manually created dataset with 30 instances per use-case. While the most critical aspect of the architecture, the Formalizer, was evaluated, a formal evaluation of the Interpreter and the full configuration pipeline were omitted for the reason that a user study is required to evaluate these aspects. This paper demonstrates that creating a productive configuration copilot is feasible but does not study the extent to which value is provided to real users in a real-world scenario.

6.2. Future Work

To address the limitations of this paper, the configuration copilot shall be evaluated on more complex use-case from practice in a user study. The configuration copilot itself shall be extended: When a configuration as specified by the user is unsatisfiable, the configuration copilot shall suggest alternatives instead of reverting to the last satisfiable configuration. Additionally, soft constraints in the form of 'If possible, I would like to ...' shall be introduced.

References

[1] L. Zhang, Product configuration: A review of the state-of-the-art and future research, International Journal of Production Research 52 (2014) 6381–6398. doi:10.1080/00207543.2014.942012.

[2] M. Yi, Z. Huang, Y. Yu, Creating a sustainable e-commerce environment: The impact of product configurator interaction design on consumer personalized customization experience, Sustainability 14 (2022). URL: https://www.mdpi.com/2071-1050/14/23/15903. doi:10.3390/su142315903.

[3] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, C. Xiong, Codegen: An open large language model for code with multi-turn program synthesis, in: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023, OpenReview.net, 2023.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, I. Polosukhin, Attention is all you need, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 30, Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[5] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, D. Roth, Recent advances in natural language processing via large pre-trained language models: A survey, ACM Computing Surveys (2023).

[6] C. Ling, X. Zhao, J. Lu, C. Deng, C. Zheng, J. Wang, T. Chowdhury, Y. Li, H. Cui, X. Zhang, T. Zhao, A. Panalkar, W. Cheng, H. Wang, Y. Liu, Z. Chen, H. Chen, C. White, Q. Gu, C. Yang, L. Zhao, Beyond one-model-fits-all: A survey of domain specialization for large language models, CoRR abs/2305.18703 (2023). URL: https://doi.org/10.48550/arXiv.2305.18703. doi:10.48550/ARXIV.2305.18703. arXiv:2305.18703.

[7] D. Benavides, A. Felfernig, J. A. Galindo, F. Reinfrank, Automated analysis in feature modelling and product configuration, in: Safe and Secure Software Reuse: 13th International Conference on Software Reuse, ICSR 2013, Pisa, June 18-20. Proceedings 13, Springer, 2013, pp. 160–175.

[8] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, MiniZinc: Towards a standard CP modelling language, in: CP, volume 4741 of *LNCS*, Springer, 2007, pp. 529–543.

[9] A. A. Falkner, A. Haselböck, G. Krames, G. Schenner, R. Taupe, Constraint solver requirements for interactive configuration, in: L. Hotz, M. Aldanondo, T. Krebs (Eds.), Proceedings of the 21st Configuration Workshop, Hamburg, Germany, September 19-20, 2019, volume 2467 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019, pp. 65–72. URL: http://ceur-ws.org/Vol-2467/paper-12.pdf.

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems, volume 33, Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[11] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, in: Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22, Curran Associates Inc., Red Hook, NY, USA, 2024.

[12] B. Wang, Z. Wang, X. Wang, Y. Cao, R. A. Saurous, Y. Kim, Grammar prompting for domain-specific language generation with large language models, in: A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, S. Levine (Eds.), Advances in Neural Information Processing Systems, volume 36, Curran Associates, Inc., 2023, pp. 65030–65055. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/cd40d0d65bfebb894ccc9ea822b47fa8-Paper-Conference.pdf.

[13] G. Poesia, A. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, S. Gulwani, Synchromesh: Reliable code generation from pre-trained language models, in: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022, OpenReview.net, 2022.

[14] L. Pan, A. Albalak, X. Wang, W. Wang, Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning, in: H. Bouamor, J. Pino, K. Bali (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2023, Association for Computational Linguistics, Singapore, 2023, pp. 3806–3824. URL: https://aclanthology.org/2023.findings-emnlp.248. doi:10.18653/v1/2023.findings-emnlp.248.

[15] P. Kogler, A. Falkner, S. Sperl, Reliable generation of formal specifications using large language models, in: SE 2024 - Companion, Gesellschaft für Informatik e.V., 2024, pp. 141–153. doi:10.18420/sw2024-ws_10.

[16] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, G. Synnaeve, M. Ai, Code llama: Open foundation models for code, 2023. URL: https://github.com/facebookresearch/codellama.

[17] MistralAI, Codestral introduction (2024). URL: https://mistral.ai/news/codestral/.

[18] AI@Meta, Llama 3 model card (2024). URL:

https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

[19] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, W. E. Sayed, Mistral 7b, 2023. arXiv:2310.06825.

[20] M. Mendonca, M. Branco, D. Cowan, S.p.l.o.t. - software product lines online tools, In Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2009) 761–762. doi:10.1145/1639950.1640002.

[21] Gecode Team, Gecode: Generic constraint development environment, 2006. Available from http://www.gecode.org.

[22] M. LEVANDOWSKY, D. WINTER, Distance between sets, Nature 234 (1971) 34–35. URL: https://doi.org/10.1038/234034a0. doi:10.1038/234034a0.

[23] A. Falkner, A. Haselböck, G. Krames, G. Schenner, H. Schreiner, R. Comploi-Taupe, Solver requirements for interactive configuration, JOURNAL OF UNIVERSAL COMPUTER SCIENCE 26 (2020) 343–. doi:10.3897/jucs.2020.019.

[24] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, M. Stumptner, Configuring large systems using generative constraint satisfaction, IEEE Intelligent Systems 13 (1998) 59–68. URL: https://doi.org/10.1109/5254.708434. doi:10.1109/5254.708434.

[25] K. Czarnecki, S. Helsen, U. W. Eisenecker, Formalizing cardinality-based feature models and their specialization, Software Process: Improvement and Practice 10 (2005) 7–29. URL: https://doi.org/10.1002/spip.213. doi:10.1002/spip.213.