A Copilot for the Engineering of Product Configuration Knowledge

Konstantin Herud[2], Antonia Heyder[2], Julian Schubert[2] and Joachim Baumeister[1,2]

[1]University of Würzburg, Am Hubland, Würzburg, Germany
[2]denkbares GmbH, John-Skilton-Straße 8, Würzburg, Germany

Abstract

This paper describes an AI-based assistant (Copilot), that is designed to support the creation, refinement, and transformation of product configuration knowledge. We introduce practical use cases for such a Copilot system and then present a prototypical implementation of the most demanding use cases. To validate its utility, we conduct a case study centered on the automated translation of an established configuration knowledge base into COOM code. The findings highlight the Copilot's potential to reduce manual effort and thus accelerate the digitalization of configuration knowledge in industrial settings.

Keywords

Product Configuration, Code Assistance, Knowledge Translation, Large Language Models

1. Introduction

In this paper, we introduce an AI-based Copilot system designed to assist knowledge workers in their day-to-day tasks. Knowledge workers are professionals who create, maintain, and execute domain-specific knowledge as a central component of their roles. Their work often involves creativity, complex reasoning, structured knowledge representation, and continuous adaptation to evolving requirements.

A timely and illustrative example of such knowledge-intensive work is the definition and maintenance of product configuration knowledge—the key task when developing product configurators [1, 2, 3]. These systems aim to enable highly automated customization processes by relying on formally represented product knowledge. Product configuration mainly solves a constraint satisfaction problem, where configurable variables, which we refer to as *features*, must satisfy defined constraints. Constraints and calculations specifying valid feature combinations and their dependencies constitute the system's *behavioral knowledge*. Developing and maintaining such configurators remains largely manual, requiring collaboration among knowledge workers who address tasks such as:

- Defining product structures and their associated features
- Specifying behavioral knowledge that constrains valid feature combinations
- Integrating new features and updating their interdependencies
- Identifying and removing obsolete or unused knowledge elements
- Translating legacy knowledge sources into formalized representations

These activities require a deep understanding of the domain and significant cognitive effort due to the scale and complexity of the knowledge bases, which are often highly interdependent and dynamic.

To support this type of knowledge work, we present a Copilot system that assists users throughout the knowledge engineering lifecycle. The Copilot operates in the context of COOM, a domain-specific language (DSL) specifically designed for representing product configuration knowledge [4]. The system is fully integrated into KnowWE [5], a collaborative knowledge engineering environment. KnowWE supports both textual and visual editing workflows, enabling users to interact with the COOM language either directly or through abstraction layers that hide the underlying DSL syntax. Through this integration, the Copilot provides intelligent, context-sensitive support that aims to reduce the

FGWM Workshop 2025: Workshop der GI-Fachgruppe Wissensmanagement
joba@uni-wuerzburg.de (J. Baumeister)

manual workload, enhance consistency, and improve the overall efficiency and quality of the knowledge engineering process.

Related Work In light of the significant advancements in generative AI technologies, promising opportunities are also emerging in software development to optimize and support processes. AI assistants can provide intuitive support tools by offering context-based code suggestions during programming. Copilot systems aim to reduce the effort involved in routine tasks and thereby increase the efficiency and productivity of developers [6]. Today, these systems are based on powerful Large Language Models (LLM) and are capable of generating code fragments and automatically completing existing code.
A prominent example in software development is *Github Copilot* [7], which was first released in 2021 and can be integrated into various software development environments. Github Copilot is described as an assistant for *AI-Pair Programming*. This references the common practice of pair programming, where experienced and less experienced developers work together on a project. TabNine [8] is also an AI-based code completion assistant that uses machine learning to complete code in various languages. The system can be integrated into multiple development environments and supports a wide range of programming languages. Amazon CodeWhisperer [9] is another system that provides real-time code snippets during development. It is based on machine learning and has been trained on Amazon's internal codebase as well as open-source projects. Domain-specific languages (DSLs) pose a challenge for Copilot systems, as their syntax and semantics often differ from general-purpose programming languages and typically have only limited training data available [10]. Nevertheless, several approaches to implementing Copilot systems for DSLs already exist. One example is the AI-based code generation for the hardware description language Verilog, that aims to automatically complete Verilog code [11]. Here, language models pre-trained on established programming languages are fine-tuned for the specific task in order to generate syntactically correct and context-sensitive code. Another interesting example is *Seq2SQL* [12]. Instead of relying solely on existing data, the model learns via reinforcement learning using *trial and error* to translate natural language into SQL queries. Correct SQL generations provide a reward signal that the model uses to improve future decisions. This approach shows potential, as only a limited data basis is often available for DSLs. In a survey among developers, Github Copilot emerged as the most widely used Copilot system. However, the most regularly used tool was ChatGPT, highlighting the relevance of non-integrated AI assistants for code generation [13].
Retrieval-Augmented Generation (RAG) enhances LLMs by integrating external information retrieval into the generation process. Instead of relying solely on pre-trained data, RAG systems dynamically fetch relevant information from external sources—such as databases or documents—at query time. This approach improves the accuracy and relevance of generated responses, particularly in domain-specific or rapidly evolving contexts [14]. Building upon RAG, *Agentic RAG* incorporates autonomous AI agents capable of planning, decision-making, and tool utilization. These agents manage multi-step tasks, adapt retrieval strategies, and refine outputs iteratively, making them well-suited for complex workflows that require dynamic reasoning and interaction with various tools [15].
This paper builds upon existing work and integrates Agentic RAG into a Copilot focusing on the task of translating existing legacy knowledge into the current knowledge base.

Paper Structure In the following section we first introduce the COOM language, followed by a discussion of the requirements for a Copilot system for product configuration knowledge. We then present a prototype implementation of the Copilot. Subsequently, a case study involving a limited number of participants provides insights into the practical usefulness of the current implementation.

2. Product Configuration Knowledge with COOM

COOM is an open domain specific language for defining product configuration knowledge [16]. There exists an open-source tool-suite for the processing of COOM knowledge (https://github.com/potassco/coom-suite), but also proprietary implementations are available that are used in industrial applications.

2.1. Basic COOM Syntax

We introduce the core concepts of the COOM language through an illustrative example. For a comprehensive and detailed introduction, we refer to coom-lang.org. In the following, we demonstrate key features of the COOM language using examples from a coffee machine producer. Specifically, we define a `Coffee_Machine` model, where customers can configure the *grinding unit* of the coffee machine (the further components of a coffee machine were omitted due to lack of space). The knowledge base is structured hierarchically, composed of feature elements that collectively describe the configurable grinder unit. In our example, the product markup defines an instance `machine` of the type `Coffee_Machine`. This structure is recursively built from enumeration types `Grinding_Type`, `Grinder_Material`, and `Noise_Level`.

```
product {
    Coffee_Machine   machine
}

structure Coffee_Machine {
    Grinding_Type    type
    Grinder_Material material
    Noise_Level      noise_level
}

enumeration Grinding_Type    { impact_cropping  disc_cropping  cone_cropping }

enumeration Grinder_Material { steel  ceramic }

enumeration Noise_Level      { quite  medium  noisy }

behavior Coffee_Machine {
  combinations ( material type )
  allow        ( steel    (impact_cropping, disc_cropping, cone_cropping) )
  allow        ( ceramic  (cone_cropping, impact_cropping) )
}
```

In addition to terminological knowledge, the model also incorporates behavioral knowledge, which imposes constraints on the permissible values defined in the enumerations. For instance, the example includes a behavior block specifying that `steel material` machines are permitted with all three grinder types, i.e., `impact_cropping`, `disc_cropping`, and `cone_cropping`. Additionally, `ceramic` material is only allowed with the types `cone_cropping` and `impact_cropping`.

Beyond this example, the COOM language provides a rich set of expressive constructs for representing complex configuration knowledge, see [16].

2.2. COOM Development with KnowWE

KnowWE is a semantic wiki platform designed for the collaborative knowledge engineering [5]. Through a range of powerful extensions, KnowWE supports the development and maintenance of product configuration knowledge, making it particularly suited for complex, domain-specific applications. Configuration knowledge can be entered using the textual COOM syntax, as described in the previous section. At the same time, graphical editors are available to facilitate a more intuitive modeling experience. KnowWE has already proven its value in several large-scale industrial projects, where it has been used to model product configurators for complex machinery. Figure 1 shows a toy example project, where configuration knowledge of coffee machines is developed and maintained. Here, the above mentioned combinations knowledge is depicted in a graphical editor.

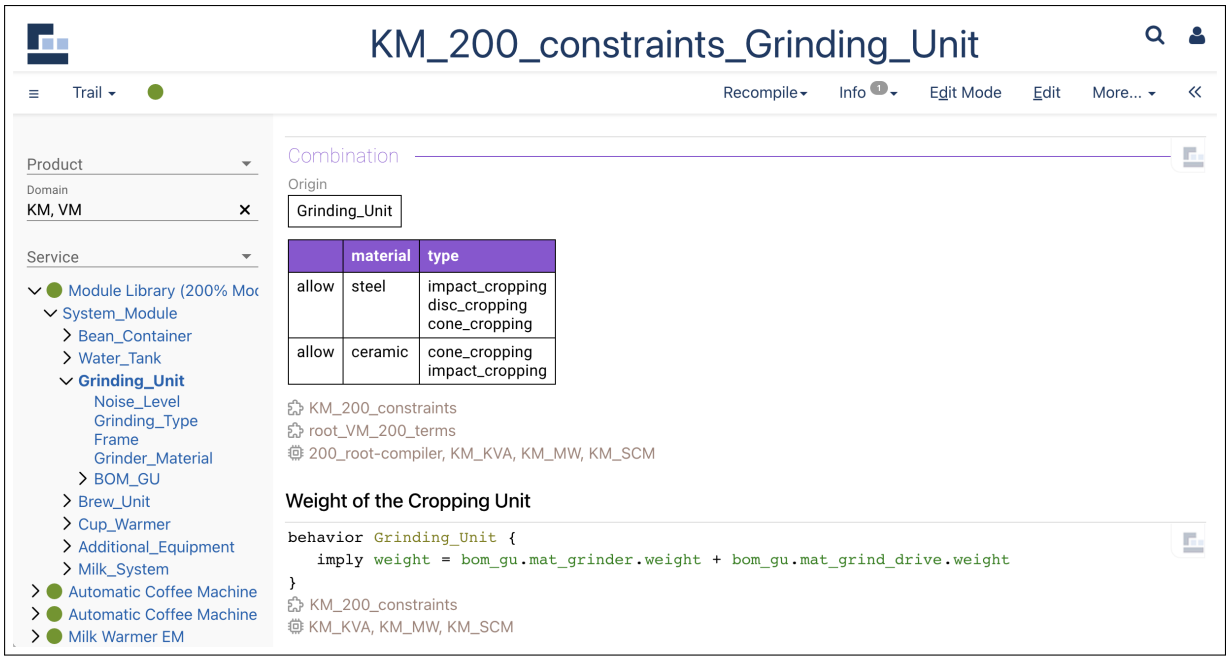

Figure 1: Product knowledge for configuring coffee machines in the tool KnowWE, defining allowed combinations for values of material and grinder type.

3. A Copilot for Product Configuration Knowledge

3.1. Requirements

As the practice-oriented basis for the development of the Copilot, we first conducted a requirements analysis. First, a catalog of relevant use cases for the use of the Copilot was developed in a design workshop with COOM experts and industrial knowledge engineers. In summary, nine core use cases where identified, including the explanation and documentation of existing knowledge, the prompt-based creation of new knowledge but also the translation of existing legacy knowledge into a new knowledge format.

A Internal Use Cases

A1: Explanation & Documentation Generation of comments for knowledge sections to improve expandability, comprehensibility, and documentation of the knowledge base.
Example Prompt: *Give me documentation for this snippet.*

A2: Dependency Analysis Identification of dependencies between various features in the knowledge base, based on constraints, derivations, and structural properties.
Example Prompt: *Which features are dependent on feature x?*

A3: Dependency Interpretation Explanation of the meaning and effects of identified dependencies on the overall system.
Example Prompt: *What do the dependencies for feature x mean?*

A4: Generation of Terminology and Constraints Creation of enumerations, structures, and constraint knowledge based on given textual descriptions. From natural language input, the relevant element type and associated properties must be inferred.
Example Prompt: *Generate an Enumeration / a Structure / a Constraint based on this description...*

A5: Test Case Generation Test cases and their values are created based on defined constraints, rules, and conditions.
Example Prompt: *Generate test cases based on these values and constraints.*

B External Use Cases

B1: Generation from Existing Code Generate COOM knowledge based on external configuration knowledge, distinguishing between structure knowledge (enumerations, structures) and constraint knowledge.
Example Prompt: *Generate an enumeration / a structure / a constraint from this given code...*

B2: External Feature Listing Listing and representation of all existing features from an external knowledge base, especially relevant during migration projects.
Example Prompt: *List all external features from...*

B3: Missing External Knowledge Identification Identification of external knowledge that is missing in the current knowledge base, important for migration projects and completeness checks.
Example Prompt: *Is there any external knowledge that has not yet been represented in COOM base?*

B4: Dependency Identification Identification and representation of dependencies between features, and finding specific features that depend on a given feature.
Example Prompt: *Which features have a dependency on feature x?*

3.2. Prototype Implementation

To evaluate the practical feasibility of our proposed approach, we implemented a prototype version of the Copilot system as an extension of the semantic wiki KnowWE. The implemented Copilot currently supports the use cases explanation (A1), generation (A4), and translation of legacy knowledge into COOM (B1). These capabilities are seen as the core interaction and automation features that enable the Copilot to assist users in typical knowledge engineering tasks. A high-level conceptual design of the prototype implementation is illustrated in Figure 2. The architecture addresses the challenge arising from an inherently simple user interface that must handle a range of complex tasks. To manage this "simple surface but extreme depth" tension, we introduced three layers:

1. **Agent Instances**. Each user request first reaches a *general-purpose agent* capable of directly resolving straightforward queries. This agent delegates more complex or specialized tasks, such as knowledge generation or legacy knowledge translation, to appropriate expert agents. Each expert has tailored capabilities and a specialized toolset optimized for its specific responsibilities.
2. **Agent Environment**. Three key components support task execution:
   - **Prompt Creation**: Dynamically assembles context for agents by integrating static elements (e.g., fixed agent instructions and few-shot examples) with dynamic elements (e.g., user requests and the current system state like the wiki page currently open).
   - **Tool Set**: Empowers agents to autonomously execute complex actions, including querying and modifying system content, documentation, compiling newly generated knowledge, and even invoking other agents. Providing tools instead of upfront retrieval ensures scalability.
   - **LLM Set**: Offers different LLMs balancing capability and performance, ranging from fast, lightweight models for summarizing content (flash models) to more capable reasoning models.
3. **System Data Sources**. The agent can access three distinct data sources using its tools:
   - **Knowledge**: Core knowledge base used and maintained within KnowWE.
   - **Documentation**: Detailed reference material supporting system components and tasks.
   - **Legacy Code**: Existing codebase, used particularly during the translation of legacy code into new knowledge.

By layering simplicity with advanced automation, the architecture supports efficient and scalable knowledge engineering workflows.

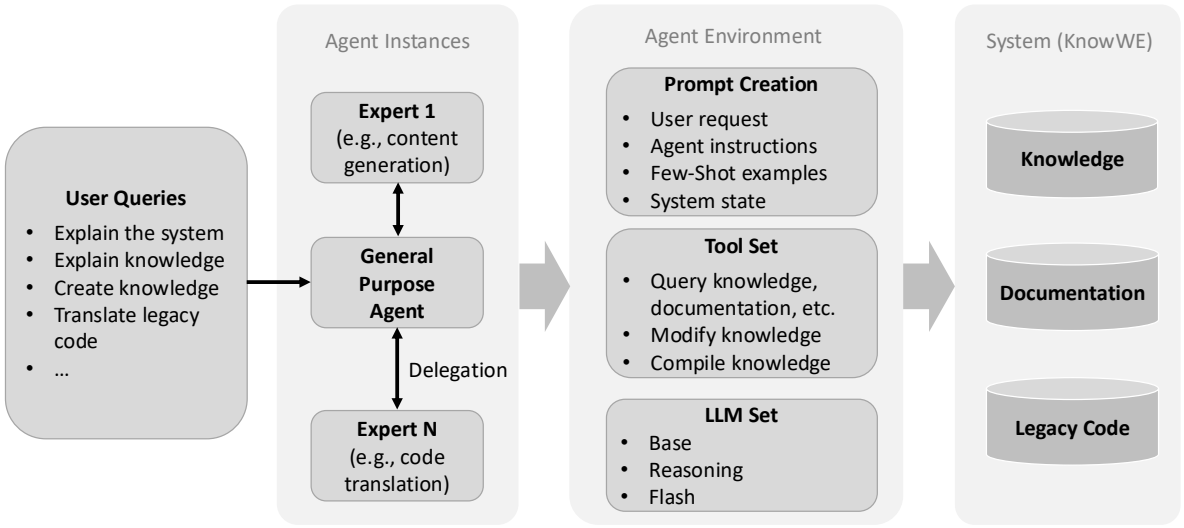

Figure 2: Conceptual overview of the Copilot agents, illustrating the flow from user queries through a general-purpose agent with expert delegation, into a structured agent environment comprising prompt creation, tool sets, and LLMs, ultimately interfacing with the KnowWE system to access or modify knowledge, documentation, and legacy code.

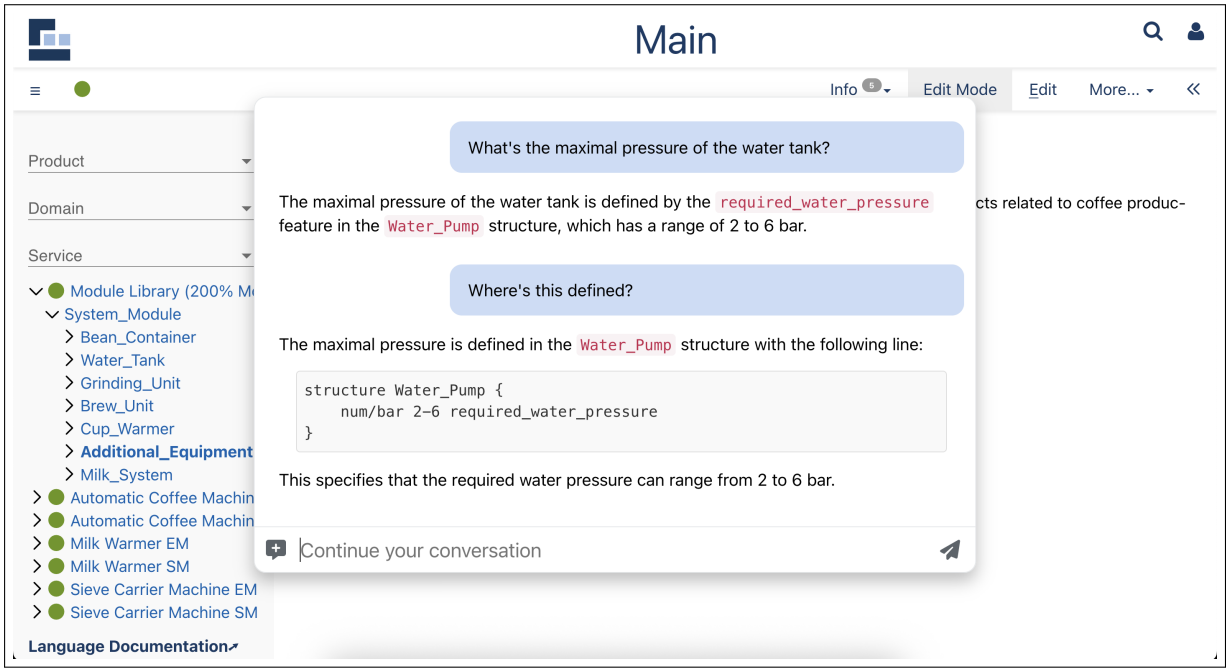

Figure 3: Copilot answers questions about the COOM knowledge base. The agent retrieves the maximal pressure of the water tank from the underlying knowledge base and outputs the relevant code snippet.

In the following we show sample usages of the Copilot. The Copilot opens as a opaque chat window of the KnowWE system, where the user can prompt queries and tasks. Figure 3 illustrates an example from the coffee machine domain, in which a user poses a series of queries directed at the knowledge base. These queries involve specific questions such as identifying the maximal pressure of the water tank and determining where the definition of this knowledge can be found. Figure 4 shows the dialog between the Copilot with the user, where a legacy snippet of configuration knowledge in Python is automatically translated into COOM language. The translation can be directly inserted into the current page for possible further adaptation and refinement.

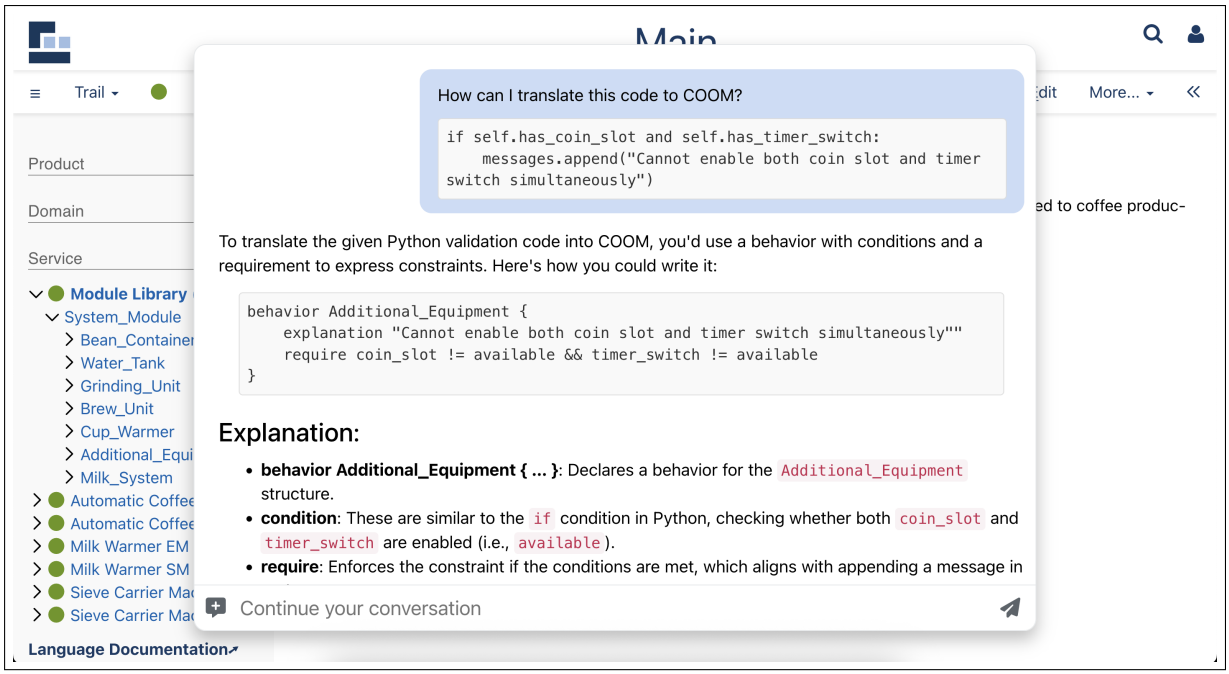

Figure 4: Copilot translating legacy configuration knowledge (in Python) into COOM language. In addition to a valid COOM code snippet, the Copilot provides an explanation of the suggested implementation.

4. Case Study

The case study investigated the extent to which the current approach of the Copilot is capable of transforming procedural knowledge into declarative COOM code. The other capabilities of the Copilot described earlier have not yet been systematically examined; however, sample interviews with knowledge engineers show very positive feedback. The case study aimed to provide insights into the practical utility and the post-processing effort required for the generated results (COOM code).

4.1. Experimental Setup

Participants

Recruiting a sufficiently large sample of participants poses a considerable challenge. To ensure the validity of the study, participants are required to possess the necessary qualifications and, ideally, engage in the investigated task on a daily basis. Furthermore, an excessively heterogeneous level of participant competence could substantially compromise the reliability and interpretability of the findings. In total, we found 9 COOM experts to participate in the case study, including 6 employees from the company denkbares GmbH and 3 industrial knowledge workers in the domain of machinery engineering. Prior research in domain-specific evaluations indicates that even small samples can yield valuable qualitative insights [17].

Of the 9 participants, 7 identified as male and 2 as female. The participants' ages ranged from 24 to 43 years ($M = 31.33$, $SD = 6.60$). Most participants have between 4 and 10 years of experience in software development. The majority have 1 to 3 years of experience with COOM. The use of AI assistance systems varies considerably, with the majority of participants using such systems on a daily basis. Two participants reported using AI assistants more than five times a day, whereas one person uses them less than once a month.

Pre-Task Data Collection and Usefulness Assessment

Before completing the tasks, data on demographics, prior experience, and the previously perceived overall usefulness of the Copilot where collected. Overall usefulness was measured using the *Perceived*

Usefulness scale from the *Technology Acceptance Model* (TAM) [18]. This questionnaire was administered both before and after the task. It was especially also presented before the task, to capture participants' initial expectations regarding an AI assistant in this domain before they see its actual performance.

The participants went through a tutorial for COOM, KnowWE, and the Copilot and they solved small practical tasks. This created a common basis for the main task of the case study. The tutorial consisted of a theoretical introduction and a practical exercise. In the practical part, participants were asked to write simple COOM code themselves. This served as a check that they had understood and could apply the concepts, thus ensuring a common knowledge basis across all participants.

Main Task Procedure

Participants were given generated COOM code, which they must review for errors and correct them directly in KnowWE as needed. This part was audio recorded. They were asked to verbalize all their thought processes aloud following the *Thinking Aloud* method [19]. After reviewing an introductory page describing the structure of the tasks, they begun the task.

The main task consisted of examining 6 generation results produced for predefined benchmarks. Benchmarks were derived from an existing industrial knowledge base by extracting representative COOM modeling problems and adapting them to a manageable scope while preserving their original characteristics. We included two benchmark types: enumeration and behavior. Participants validated the Copilot's generation results for each benchmark by checking the syntactic correctness of the code and assessing whether it satisfied the given modeling problem. In cases where the output was incorrect or incomplete, participants noted the necessary corrections. The order of the benchmarks was partially controlled: enumeration benchmarks were always presented first, followed by behavior benchmarks. The tasks were carried out entirely in KnowWE.

KnowWE provides support tools such as syntax highlighting, auto-completion, and access to the existing product structure. The environment is tailored to the prior knowledge of typical users; however, help resources, such as a central link to the product context, were available for less experienced users. The basis for completing the task was the original legacy code, which was shown alongside the generated COOM code. Using this contextual information, participants analyzed the generated COOM code and corrected it if necessary directly in KnowWE. After completing the revision, a reference solution was made available for comparison. There was no time limit; however, participants were encouraged to proceed efficiently, similar to a real work context.

Post-Task Evaluation

After each task, participants estimated the post-processing effort and rated the usefulness of the generated code they worked on. For each task, Error Cost, Solution Cost, and Repair Cost were used to operationalize the real user-perceived post-processing effort. Conceptually, costs reflect the cognitive and mechanical effort required to handle errors in generated code. Intuitively, these costs are defined as follows:

Error Cost describes the intellectual effort required to identify errors occurring in the generated code ("1 very easy" — "5 very complex").

Solution Cost describes the intellectual efforts to find an appropriate target translation of the legacy code, i.e., the correct COOM code ("1 very easy" — "5 very complex").

Repair Cost refers to the purely mechanical execution of the necessary modifications until all identified faults have been corrected ("1 very easy" — "5 very complex").

These are rated on a 5-point Likert scale ranging from minimal to significant effort. If participants identified errors, all parameters were assessed; for error-free code, only Error Cost was recorded. For samples where the Copilot output was correct, a cost of zero was used for the solution and repair cost.

Final Evaluation and Debriefing

After completing all tasks, participants again completed the perceived usefulness. In addition, we administered two single item judgments tailored to our context on the same scale: "I find the COOMPilot helpful." and "I would like to use the COOMPilot in the future." The repeated usefulness questions before and after the experiment gave us the opportunity to check whether the case study had led to a change in the estimated usefulness of the participants.

Participants may also have provided additional feedback, suggestions, or identified potential problems through open-ended questions. A debriefing and opportunity to discuss further questions concluded the session. The time taken per task was retrospectively determined using the audio recordings.

4.2. Results

A descriptive and qualitative analysis was carried out without inferential statistical evaluation due to the small number of participants. The focus was on the practical usefulness of the generated code.

Error, Solution, and Repair Costs

Figure 5 shows the result values of the expert ratings of the follow-up work for the generated COOM code. In the overall consideration of Error Cost, the average effort value is 2.15 (*SD* = 0.82). The mean

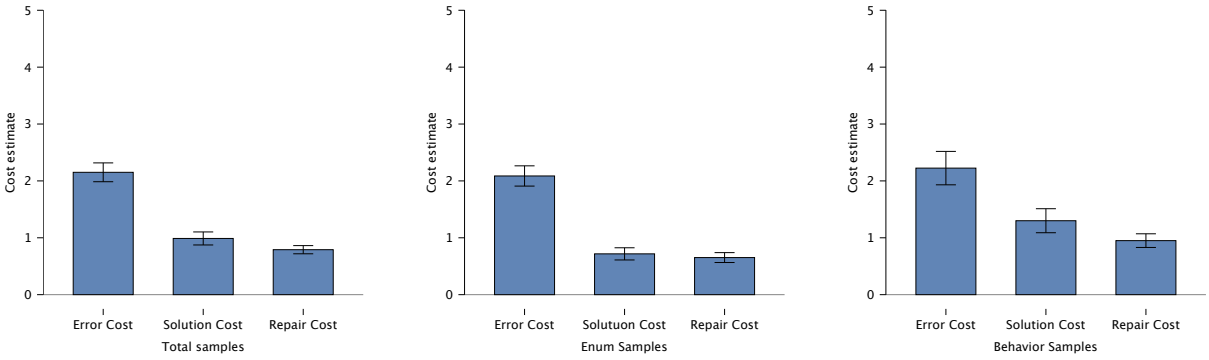

Figure 5: Experts' follow-up work based on Error, Solution and Repair Costs.

Solution Cost amounts to 0.99 (*SD* = 1.28), while the Repair Cost reaches an average effort value of 0.79 (*SD* = 1.03). For **enumerations**, the Error Cost according to expert assessment averages an effort value of 2.09 (*SD* = 0.70). On average, the Solution Cost was rated at an effort value of 0.72 (*SD* = 1.03). The average effort value for the Repair Cost is 0.65 (*SD* = 0.98). For **behaviors**, the Error Cost was reported with an average effort value of 2.23 (*SD* = 0.95). The Solution Cost shows an average effort value of 1.30 (*SD* = 1.47), and the Repair Cost was rated on average at 0.95 (*SD* = 1.09). Scores lower than one stem from the fact that for correct answers of the Copilot the Solution and Repair cost were set to zero, reflecting that once the solution is identified as correct, no more work needs to be done by the knowledge worker for this sample.

Perceived Usefulness and Required Time

A total of 54 ratings were collected for the **perceived overall usefulness**, based on 9 participants and six usefulness items each from the TAM. Before the evaluation, a *perceived usefulness* of 6.09 (*SD* = 1.69) was recorded on a scale from 1 to 7, and 6.07 (*SD* = 1.24) after the evaluation. For the *perceived usefulness per benchmark*, assessed on a scale from 1 to 5, an overall mean value of 4.44 (*SD* = 0.66) was observed. For enumerations, the average perceived usefulness per benchmark is 4.44 (*SD* = 0.74), and for behaviors it is on average 4.45 (*SD* = 0.56). The statement *"I would like to use the Copilot in the future."*, rated on a

scale from 1 to 7, achieved an average usefulness value of 6.44 (*SD* = 1.07). The item *"I find the Copilot helpful"* reached an average usefulness value of 6.56 (*SD* = 0.96).

The **average post-processing time** for enumerations is 1:33 minutes (*SD* = 0:52), and for behaviors it is on average 3:16 minutes (*SD* = 1:53). Across both COOM types (enumeration and behavior), the mean post-processing time is 2:21 minutes (*SD* = 1:40). In previous interviews some of the participants reported, that the total time duration for creating behaviors from legacy sources is approximately 30 minutes.

Qualitative Findings

The qualitative evaluation, based on Thinking Aloud and open feedback, provides further practical insights into the validity of the *Error Score* and the usefulness of the Copilot. Small errors affecting only a few tokens were described as more difficult to identify. Regarding the Repair Cost, substitutions were sometimes perceived—similar to deletions—as minimally effortful.

The Copilot was described as helpful despite the necessary post-processing. In particular, it was noted that even erroneous results offered potential for several reasons. The provision of the basic code structure reduced typing effort and provided cognitive impulses, helping users to stay in the flow. The availability of retrieval results also reduced the need for additional research, which was seen as a significant relief. Long-term experts viewed the post-processing somewhat more critically, as they desired complete automation.

For a future system, participants suggested the ability to query reasoning aspects in order to receive explanations for Copilot outputs. Alternative code suggestions and linking with test cases were also mentioned. Furthermore, suggestions included assessments of the safety of generated results and the implementation of a feedback loop. This would allow the model to learn from mistakes and continuously improve. Excessive trust in the generated code was highlighted as a potential risk factor.

4.3. Discussion

The results highlight that mostly helpful and usable code is generated, providing a solid foundation for developers. Nevertheless, manual review and potential corrections are always necessary, as occasional errors are to be expected. The usefulness of the code snippets and the overall system was rated positively, indicating that a Copilot could represent a substantial improvement over the status quo.

Novice and intermediate users saw great potential in using the Copilot, particularly for repetitive tasks and as a starting aid. This was particularly underscored by the fact that participants expressed a willingness to actually use the Copilot. In addition, the subjective post-processing effort was largely rated as low, making it clear that a significant portion of the required work could already be adequately handled by the Copilot. This aligns with previous research indicating that less experienced users especially benefit from AI assistance. Long-term experts expressed greater skepticism about the ongoing necessity of post-processing but remained generally optimistic. Referring again to the concept described by Barke et al. [20], it becomes evident that both acceleration and exploration of ideas are pertinent for experts. In this context, acceleration refers to the expedited progression of research tasks. Conversely, exploration involves the open-ended investigation of novel or less-understood topics. Constructive suggestions—such as a reasoning function or linking to test cases—highlight that experts recognize promising prospects for a Copilot and would welcome the technology in practice.

Overall, these findings suggest that Copilots are capable of supporting knowledge engineers in transforming procedural source code into declarative constraint languages. Although fully correct results cannot yet be expected, the fundamental transformation of the knowledge already appears to offer substantial workload relief and efficiency gains. This can reduce manual effort and provide a valuable foundation for further development.

5. Conclusions and Outlook

This work introduced a Copilot implementation supporting knowledge workers in the application field of product configuration. The introduced Copilot system demonstrates how AI can effectively assist knowledge workers. By automating key tasks such as documentation, dependency identification, and code generation, the assistant can significantly streamline the development and migration of configuration knowledge into structured formats like COOM. The presented case study only focussed on the use case of generation legacy knowledge into COOM code. Results of a (limited) user study demonstrate the practical benefits. Overall, the Copilot represents a promising step toward intelligent, AI-driven support tools for configuration engineering and knowledge formalization.

In the future, the Copilot system will be extended to support the remaining (and possibly further) use cases described in Section 3.1. Additionally, intent-based routing and knowledge extraction methods can be explored, allowing the Copilot to reliably output information gathered across the knowledge base [21]. Furthermore, a more comprehensive case study encompassing a broader range of use cases is necessary to fully assess the potential of Copilot systems within knowledge engineering environments. For the measurement of perceived usefulness, we adopted items from the Technology Acceptance Model (TAM). While TAM is an established model for technology acceptance, it has been superseded by the Unified Theory of Acceptance and Use of Technology (UTAUT) [22], which integrates and extends TAM. Future work should consider UTAUT as a broader theoretical foundation.

Acknowledgments

This work was partly funded by German Federal Ministry for Economic Affairs and Energy by ZIM BMWK grants KK5394901GR1 (ISCO) and KK5394902GR4 (PANKO).

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT in order to: Grammar and spelling check. After using this service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen, Knowledge-based Configuration: From Research to Business Cases, 1 ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.

[2] J. Baumeister, K. Herud, M. Ostrowski, J. Reutelshöfer, N. Rühling, T. Schaub, P. Wanko, Towards industrial-scale product configuration, in: C. Dodaro, G. Gupta, M. V. Martinez (Eds.), Logic Programming and Nonmonotonic Reasoning, Springer Nature Switzerland, Cham, 2025, pp. 71–84.

[3] A. Falkner, G. Friedrich, A. Haselböck, G. Schenner, H. Schreiner, Twenty-five years of successful application of constraint technologies at siemens, AI Magazine 37 (2017) 67–80. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/2688. doi:10.1609/aimag.v37i4.2688.

[4] J. Baumeister, K. Herud, L. Ley, J. Reutelshoefer, A framework for categorizing product configuration systems, in: LWDA'24: Lernen, Wissen, Daten, Analysen, 2024.

[5] J. Baumeister, J. Reutelshoefer, F. Puppe, KnowWE: A semantic wiki for knowledge engineering, Applied Intelligence 35 (2011) 323–344. URL: http://dx.doi.org/10.1007/s10489-010-0224-5.

[6] N. Friedman, Introducing GitHub Copilot: your AI pair programmer, 2021. URL: https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/.

[7] GitHub, GitHub Copilot, 2024. URL: https://github.com/features/copilot/.

[8] TabNine, The AI code assistant tailored to your team, 2024. URL: https://www.tabnine.com/.

[9] Amazon Web Services, Amazon CodeWhisperer - AI powered code recommendations, 2024. URL: https://aws.amazon.com/codewhisperer/.

[10] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, J. Wang, On the effectiveness of large language models in domain-specific code generation, in: ACM Transactions on Software Engineering and Methodology, 2024. URL: http://arxiv.org/abs/2312.01639. doi:10.48550/arXiv.2312.01639. arXiv:2312.01639 [cs].

[11] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, A Deep Learning Framework for Verilog Autocompletion Towards Design and Verification Automation, arXiv preprint arXiv:2304.13840, 2023. doi:10.48550/arXiv.2304.13840.

[12] V. Zhong, C. Xiong, R. Socher, Seq2SQL: Generating structured queries from natural language using reinforcement learning, CoRR abs/1709.00103 (2017). URL: http://arxiv.org/abs/1709.00103. arXiv:1709.00103.

[13] J. T. Liang, C. Yang, B. A. Myers, A large-scale survey on the usability of AI programming assistants: Successes and challenges, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024. URL: https://doi.org/10.1145/3597503.3608128. doi:10.1145/3597503.3608128.

[14] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al., Retrieval-augmented generation for knowledge-intensive NLP tasks, Advances in Neural Information Processing Systems 33 (2020) 9459–9474.

[15] A. Singh, A. Ehtesham, S. Kumar, T. T. Khoei, Agentic retrieval-augmented generation: A survey on agentic RAG, arXiv preprint arXiv:2501.09136 (2025).

[16] coom-lang, https://www.coom-lang.org, 06.2025.

[17] D. S. Young, E. A. Casey, An Examination of the Sufficiency of Small Qualitative Samples, Social Work Research 43 (2018) 53–58. URL: https://doi.org/10.1093/swr/svy026. doi:10.1093/swr/svy026, _eprint: https://academic.oup.com/swr/article-pdf/43/1/53/27826186/svy026.pdf.

[18] F. D. Davis, Perceived usefulness, perceived ease of use, and user acceptance of information technology, MIS Quarterly 13 (1989) 319–340. URL: https://www.jstor.org/stable/249008. doi:10.2307/249008, publisher: Management Information Systems Research Center, University of Minnesota.

[19] J. Nielsen, Estimating the number of subjects needed for a thinking aloud test, International Journal of Human-Computer Studies 41 (1994) 385–397. URL: https://www.sciencedirect.com/science/article/pii/S1071581984710652. doi:https://doi.org/10.1006/ijhc.1994.1065.

[20] S. Barke, M. B. James, N. Polikarpova, Grounded Copilot: How programmers interact with code-generating models, Proc. ACM Program. Lang. 7 (2023). URL: https://doi.org/10.1145/3586030. doi:10.1145/3586030.

[21] J. Schubert, M. Krug, J. Baumeister, Knowledge retrieval with LLMs using context-specific intent and slot classification, in: LWDA'24: Lernen, Wissen, Daten, Analysen, 2024.

[22] V. Venkatesh, M. G. Morris, G. B. Davis, F. D. Davis, User acceptance of information technology: Toward a unified view, MIS Quarterly 27 (2003) 425–478. URL: http://www.jstor.org/stable/30036540.