An extended experimental evaluation of an answer set programming solution to the task assignment problem*

Franz Wotawa[1,*]

[1]Graz University of Technology, Institute of Software Engineering and Artificial Intelligence, Inffeldgasse 16b/2, A-8010 Graz, Austria

Abstract

Configuring computing networks requires assigning tasks to computing nodes so that various constraints are met, such as not exceeding available memory or the maximum number of tasks a node can handle. The task assignment problem is known to be related to the bin packing problem for which efficient approximation solutions exist. In this paper, we elaborate on an answer set program solution and, in particular, on an experimental evaluation specifically focusing on transitions between satisfiable and unsatisfiable instances of the problem. We show how constraints that allow us to distinguish satisfiable from unsatisfiable instances affect overall execution time. Furthermore, we present experiments on the corresponding optimization problem, where we search for solutions that minimize the number of computing nodes used. The experiments provide practitioners with insights into applying knowledge-based and declarative solutions to the task assignment problem in practice.

Keywords

Task assignment problem, Experimental evaluation, Knowledge-based configuration

1. Introduction

Assigning tasks to computing nodes is an important problem in various domains, including the automotive industry [1], or time-triggered networks [2, 3]. More recently, authors have suggested using answer set programming (ASP) [4, 5] to solve the task assignment problem. Wotawa [6] presented an ASP model, along with an initial study showing that ASP can yield smaller instances of the problem. Furthermore, Wotawa and colleagues [7] extended the model to account for requirements such as communication costs and increased fault tolerance, while considering replication. The use of ASP for configuration and similar tasks is not new. Abseher and colleagues [8] presented an ASP model for shift scheduling. Mishra [9] used ASP for implementing a product configurator. Other, more recent work includes [10, 11].

The task assignment problem, which is related to the well-known knapsack problem [12, 13] and the bin-packing problem [14] for which approximation algorithms exist [15], comprises tasks having memory requirements to computing nodes providing memory. Let us illustrate the problem using 3 computing nodes n_1, n_2, n_3, each offering 3 kilobytes of memory, and 4 tasks t_1 to t_4 requiring 1, 1, 2, and 2 kilobytes of memory, respectively. Obviously, when assigning t_1 and t_3 to n_1, and t_2 and t_4 to node n_2, all memory requirements are fulfilled. Figure 1 shows a graphical representation of this problem. For this problem instance, there are, of course, many other solutions. The figure depicts only three of them. Depending on certain optimality criteria, we might prefer one solution over the other. In Figure 1, solutions 1 and 2 are optimal solutions when minimizing the number of used nodes in the assignment. In this example, there is no solution that consists of only one node. Note that there must not always be a solution. If we have only one computing node, n_1, the memory requirements of the 4 tasks cannot be met. Hence, we have satisfiable and unsatisfiable problem instances of the task assignment problem.

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany

*The work was supported by the Austrian Science Fund (FWF) Cluster of Excellence Bilateral AI under contract number 10.55776/COE12.

*Corresponding author.

wotawa@tugraz.at (F. Wotawa)

0000-0002-0462-2283 (F. Wotawa)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

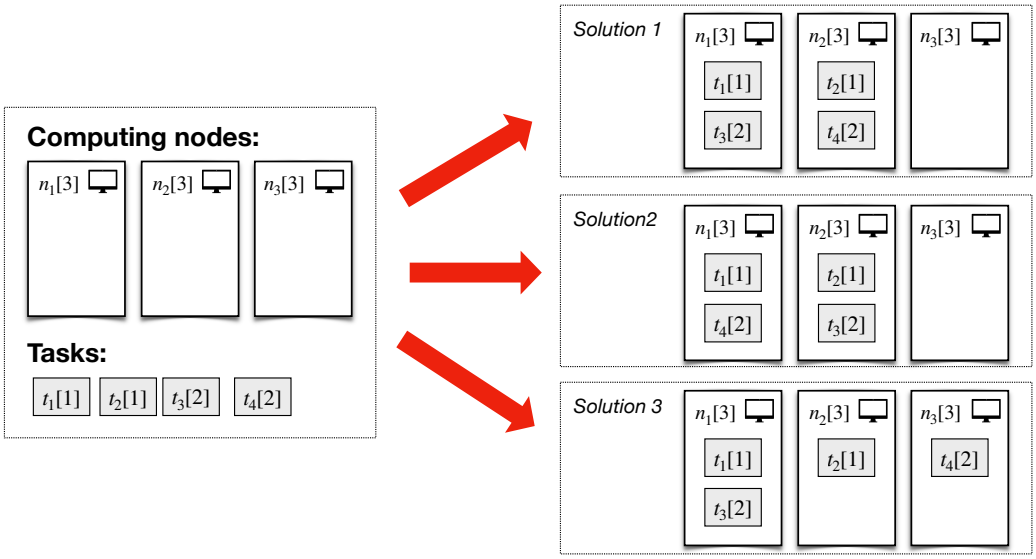


Figure 1: An example of the task assignment problem including three solutions

In this paper, we extend previous work of Wotawa [6] and provide a detailed experimental evaluation of the ASP formalization of the task assignment problem. In particular, we focus on the boundary between satisfiable and unsatisfiable instances, compare the runtime execution of a minimized ASP solution with that of a solution using global constraints to find unsatisfiable problems as early as possible, and also extend the work by introducing optimization and its impact on runtime execution. The objective of the extended experimental evaluation is to provide evidence on whether an ASP solution can be used efficiently in practice and under which assumptions. In particular, someone might be interested in obtaining a boundary such that solutions can be computed using ASPs with a high percentage. Such guarantees are important for practical applications, especially in real-time, fault-tolerant, or safety-critical systems.

It is worth mentioning that the problem of finding transitions in search spaces where problems become hard to solve has already been of interest. Cheesman and colleagues [16] showed that hard problems occur when a parameter reaches a critical value. Mitchell et al. [17] reported that the right distribution of instances and appropriate parameter values makes checking for satisfiability of random instances hard. This paper contributes to this research direction by providing evidence for such a transition between instances of the task assignment problem.

We organize this paper as follows: To be self-contained, we start by discussing the foundations behind the task assignment problem, including its optimization case. Afterward, we introduce the underlying ASP models. Section 3 comprises the experimental evaluation, including a description of the setup, the generation of the data, and the results obtained. In Section 4, we discuss the experimental results, and finally, we conclude the paper.

2. Foundations

In this section, we discuss the foundations of the task-to-computing-node assignment problem (or task assignment problem, for short), and extensions such as finding optimal solutions and considering reconfiguration. We begin by defining the task assignment problem using modified formal definitions from a previous publication [6]. For this purpose, we assume that we have k computing nodes $\mathbf{N} = n_1, \ldots, n_k$ and n tasks $\mathbf{T} = t_1, \ldots, t_n$ to be assigned to the nodes. For each node $n \in \mathbf{N}$, we know the maximum number of tasks $c(n)$ it can hold and the available memory $m(n)$. For each task $t \in \mathbf{T}$, we know its memory consumption $m(t)$. From this knowledge, we formulate several constraints that any

valid solution to the task assignment problem must satisfy.

The **task assignment problem** in general can be formulated as follows:

Given: A tuple $(\mathbf{N}, \mathbf{T}, m, c)$ where $\mathbf{N}$ is set of computing nodes, $\mathbf{T}$ is a set of tasks, $m : \mathbf{N} \cup \mathbf{T} \mapsto \mathbb{N}_0$ is a function specifying the available memory of a node and the needed memory of a task, and a function $c : \mathbf{N} \mapsto \mathbb{N}_0$ specifying the number of tasks a node can hold.

Wanted: A valid assignment of every task to a node, such that no memory or number of nodes constraint is violated. A valid assignment is one where every task $t \in \mathbf{T}$ is assigned to exactly one node $n \in \mathbf{N}$.

This task assignment problem, which is NP-hard, is an extension of the knapsack problem [12, 13] and similar to the bin-packing problem [14]. In contrast to the former, the problem involves more than one entity storing items (i.e., tasks), and, unlike the latter, we also have restrictions on the number of tasks a node can take.

For providing the constraints and the solution of a task assignment problem, we assume a function $assigned : \mathbf{N} \mapsto 2^{\mathbf{T}}$ that returns a set of tasks that is assigned to a node n. Using $assigned$, we introduce several constraints that any solution must fulfill.

First, the required memory from the tasks assigned to a node shall never exceed the available memory of this node. These constraints can be formalized as follows:

$$\forall n \in \mathbf{N} : \left(\sum_{t \in assigned(n_i)} m(t) \leq m(n) \right) \tag{1}$$

Second, the number of tasks assigned to a node shall never exceed the maximum number of tasks the node can hold, i.e., formally, we write:

$$\forall n \in \mathbf{N} : (|assigned(n)| \leq c(n)) \tag{2}$$

Furthermore, a valid solution to the task assignment problem is an assignment of every task to exactly one node, with no task assigned to two different nodes. Hence, the following constraints must hold, too:

$$\forall t \in \mathbf{T} : \exists n \in \mathbf{N} : t \in assigned(n) \tag{3}$$

$$\forall n, n' \in \mathbf{N}, n \neq n' : assigned(n) \cap assigned(n') = \emptyset \tag{4}$$

Obviously, there are task-assignment problems for which no solution satisfies all constraints. For example, if the number of tasks exceeds the number of free slots of the nodes or if the required memory for the tasks is not provided, there is no solution. Hence, we may add more constraints like the following two, describing necessary conditions that must hold for obtaining a solution:

$$|\mathbf{T}| \leq \sum_{n \in \mathbf{N}} c(n) \tag{5}$$

$$\sum_{t \in \mathbf{T}} m(t) \leq \sum_{n \in \mathbf{N}} m(n) \tag{6}$$

Equations 5 and 6 are required to find a solution, but not necessary for stating the task assignment problem.

Example: The task assignment problem instance from Figure 1 can be formulated as follows: $\mathbf{N} = n_1, n_2, n_3$, $\mathbf{T} = t_1, t_2, t_3, t_4$ with $c(n_1) = c(n_2) = c(n_3) = 4$ (meaning that there are no limitations regarding the tasks to be assigned to a node), $m(n_1) = m(n_2) = m(n_3) = 3$, and $m(t_1) = m(t_2) = 1$, $m(t_3) = m(t_4) = 2$. A solution is $assigned(n_1) = t_1, t_3$, $assigned(n_2) = t_2, t_4$, $assigned(n_3) =$.

In practice, the task assignment problem might be formulated as an optimization problem, where we search for a solution that is minimal with respect to a given evaluation function f, which maps solutions to values. For example, we might be interested in finding solutions that minimize the number of nodes where tasks are assigned. In this case, we define f as follows for a solution S, which is given by its corresponding $assigned$ function:

$$f(S) = |\{n | assigned_S(n) \neq \emptyset \wedge n \in \mathbf{N}\}| \tag{7}$$

Using $f(.)$ we define the **task assignment optimization problem**:

Given: A tuple $(\mathbf{N}, \mathbf{T}, m, c)$, where $\mathbf{N}$ is set of computing nodes, $\mathbf{T}$ is a set of tasks, $m : \mathbf{N} \cup \mathbf{T} \mapsto \mathbb{N}_0$ is a function specifying available memory need needed memory, and a function $c : \mathbf{N} \mapsto \mathbb{N}_0$ specifying the number of tasks a node can hold. A function f mapping solutions to a natural number.

Wanted: A valid assignment S of every task to a node, such that no memory or number of nodes constraint is violated, like for the task assignment problem where S is minimal with respect to the optimization function f.

Example: The solution for the task assignment problem depicted in Figure 1 $assigned(n_1) = t_1, t_3$, $assigned(n_2) = t_2, t_4$, $assigned(n_3) =$ is a minimal one considering $f(.)$ from Equation 7.

In the next subsection, we introduce the ASP implementation of the task assignment problem.

2.1. An ASP implementation of the task assignment problem

The ASP implementation in *clingo* [18, 19] syntax implements the problem description and the constraints straightforwardly. We represent a node n from $\mathbf{N}$ as predicate `node(`n`)`, and a task t from $\mathbf{T}$ as predicate `task(`t`)`. We specify the task capacity c of a node n using the predicate `tcapacity(`n`,`c`)`, the memory m of nodes as `mcapacity(`n`,`m`)`, and the memory requirement r of task t as `memory(`t`,`r`)`.

The following ASP code implements searching for solutions to a given task assignment problem:

```
1  % Generate a selection of a node for each task
2
3  { select(T,N) : node(N) } = 1 :- task(T).
4
5  % Constraints
6
7  % No 1: Do not exceed the maximum number of tasks
8
9  noTasksAssigned(M,N) :- M = #count { T : select(T,N)}, node(N).
10
11 :- noTasksAssigned(M,N), tcapacity(N,C), M>C.
12
13 % No 2: Do not exceed the maximum memory of a node
14
15 memRequired(M,N) :- M=#sum{NM,T: select(T,N),memory(T,NM)}, node(N).
16
17 :- memRequired(M,N), mcapacity(N,C), M > C.
18
19 % No 3: Global constraints
```

```
20
21 totalCapacity(C) :- C=#sum{T,N:tcapacity(N,T) }.
22 totalNrTasks(C) :- C=#count{T:task(T) }.
23 totalMemReq(C) :- C=#sum{M,T:memory(T,M)}.
24 totalMem(C) :- C=#sum{N,CN:mcapacity(CN,N)}.
25
26 :- totalCapacity(C), totalNrTasks(TC), C < TC.
27 :- totalMemReq(Ctask), totalMem(Cnode), Ctask > Cnode.
```

In the first part of the implementation, we generate all potential solutions (Line 3). Afterward, we add the constraints capturing not exceeding the maximum number of possible tasks per node (lines 9–11), not exceeding the available memory of a node when assigning tasks (lines 15–17), and the global constraints enabling us to find cases where no solution exists as early as possible in lines 21–27.

The optimization variant of the implementation is equivalent to the previous ASP program, including the following extension:

```
1 % Compute the total number of nodes with at least one task assigned
2
3 totalNoUsedNodes(C) :- C=#count{N:select(T,N),task(T)}.
4
5 % Minimize the used nodes.
6
7 #minimize {N:totalNoUsedNodes(N)}.
```

This extension computes the total number of nodes in Line 3 that have at least one assigned task. The final *clingo* statement in Line 7 minimizes this number.

3. Experimental evaluation

With the experimental evaluation, we want to investigate boundaries of application as well as gaining more information regarding potential issues at the borderline between satisfiable and unsatisfiable instances of the task assignment problem. Furthermore, we want to extend our evaluation to optimization problems and their effect on runtime. In the following, we discuss the setup and the results obtained. For each part of the result section, we discuss the generation of example instances.

3.1. Setup

All experiments discussed in this paper were carried out on a MacBook Pro with an Apple M4 Pro processor, 48 GB of main memory, and macOS Tahoe 26.3.1. During the experiments, the computer was used for writing and other similar tasks, which should not have had a severe impact on runtime.

For conducting the experiments, we used the ASP solver *clingo* version 5.7.1, which implements the 64-bit address model. We did not change any standard configuration nor optimize any parameters of *clingo* during the evaluation, which might have affected the results.

All example instances are generated using a Java program. The instances were generated randomly, considering parameters such as the number of nodes and tasks, as well as the memory requirements and the available memory. We discuss these parameters, as well as the number of generated instances for each experimental setup, separately. We used a Python program for further analysis of the obtained results, i.e., mainly to compute minimum, maximum, mean, and median values for the different parameters. Note that in the experiments, we did not use the maximum number of assigned tasks for each node. We set this value for all computing nodes to the maximum number of tasks in an example instance.

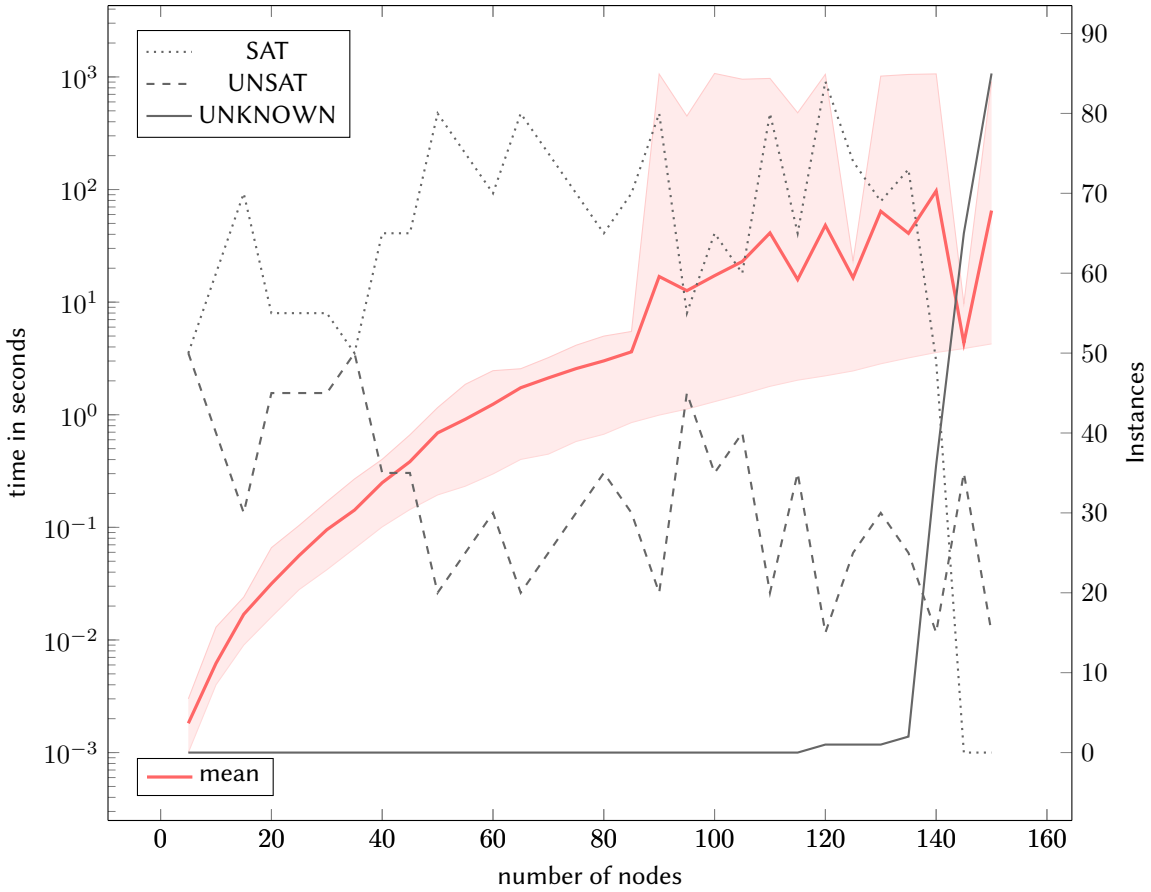


Figure 2: Mean solving runtime for 5 to 160 nodes and a corresponding number of tasks $t = 2.9 \cdot n$. Besides the minimum, maximum, and mean runtime in seconds, this figure also provides information regarding the number of SAT, UNSAT and UNKOWN instances.

3.2. Results

In the following, we describe the different experiments, their specific setup and data, and the obtained results. We start with an initial experiment that captures a case where the satisfiable and unsatisfiable solutions are represented in a 2-to-1 ratio on average. In the succeeding experiment, we focus on the transition phase between satisfiable and unsatisfiable problem instances. For this purpose, we fix the number of nodes and vary the task. Afterward, we clarify the question of the impact of the global constraints on the execution time. Finally, we conduct an experiment focused on optimization, using the same data as in the initial experiment to ensure comparability.

3.2.1. Initial experiment

The objective of the first experiment is to clarify the runtime of solving a problem depending on the number of nodes n, where the number of tasks t is set to $t = n \cdot \frac{29}{10}$. This equation was experimentally determined to ensure that the ratio of satisfiable (SAT) to unsatisfiable (UNSAT) problem instances is about 2 to 1, assuming 0 to 10 available memory per node and 0 to 4 required memory units per task. The number of tasks allowed per node was fixed to ensure this value does not affect satisfiability. For the experiment, we generated 100 instances with the number of computing nodes ranging from 5 to 150, with a step size of 5. We further set the time boundary for finding solutions using *clingo* to 30 seconds. All runs were carried out 5 times using the same problem instances.

Figure 2 depicts the mean, minimum, and maximum runtime obtained. Moreover, this figure comprises the number of SAT, UNSAT, and UNKNOWN instances. An instance is set to UNKNOWN if *clingo*

was not able to provide a solution within the given time boundary. From the figure, we see that (with the two exceptions) the number of SAT instances is larger than that for UNSAT. Furthermore, starting at about 120 nodes, the number of UNKNOWN instances increases, indicating that the likelihood of requiring more than 30 seconds to provide a solution increases.

It is worth noting that the number of UNKNOWN instances increases significantly after 135 nodes, whereas the number of SAT instances drops to 0; this does not hold for UNSAT. Hence, it seems that finding SAT solutions takes increasing time, whereas UNSAT solutions are found quickly. Hence, there is a need to conduct further experiments focusing on the boundary between SAT and UNSAT, which we discuss in the following subsection.

Note also that after about 85 nodes, the number of instances requiring a substantially longer time increases. A reason might be that problem instances and their internal *clingo* representation cause an increase in runtime. Alternatively, there might be some instances for which finding a solution is harder, and they happen to appear in instances with more than 85 nodes. Further investigations are necessary to clarify the origin.

For practical applications, requiring a certain maximum runtime of 0.1 or 1.0 seconds, we may restrict the instances to about 30 or 50 nodes, respectively.

3.2.2. The transition phase between SAT and UNSAT

After discussing the behavior of the ASP solution under a fixed ratio of nodes to tasks, we want to elaborate on the transition phase between instances that can be solved (SAT) and those that cannot (UNSAT). To avoid a far too long runtime, which may exceeding the predefined time boundary for solving of 30 seconds, we consider 25, 50, and 75 nodes, and 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120, 200, and 300 tasks, respectively, with a step size of 5. Furthermore, we consider 0 to 10 memory available per node and 0 to 5 memory required for all tasks, as in the previous experiment. We do not consider the maximum number of tasks per node. All runs were carried out 5 times using the same problem instances.

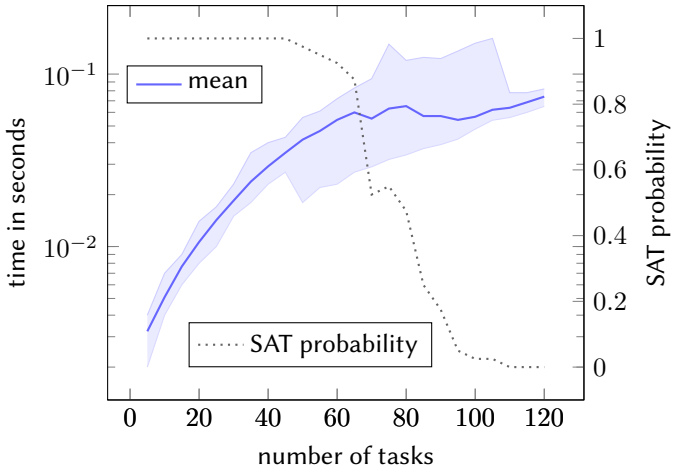
Figure 3 (a) to (c) depict the obtained results for 25, 50, and 75 nodes, respectively. In all examples, we see a boundary between SAT and UNSAT instances, and beyond it, all instances are UNSAT. When the probability of an instance decreases, the average runtime also decreases, as explained by Equations 5 and 6, allowing clingo to terminate the search early. After this boundary, the runtime again increases with the problem size. Hence, identifying UNSAT instances using necessary conditions reduces the overall runtime. Interestingly, this effect in absolute mean runtime increases with the number of nodes and, therefore, with the problem size. It is also worth noting that the transition phase in terms of tasks depends on the number of nodes, which is reasonable because for fewer computing nodes, fewer tasks are required for an instance to become unsatisfiable when the other parameters do not change.

3.2.3. Utilizing global constraints

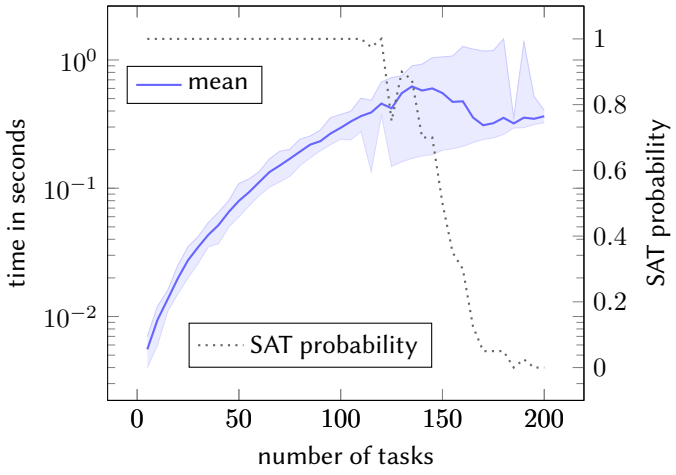
After analyzing the ordinary behavior and the runtime behavior of *clingo* when solving instances focusing on the transition from SAT to UNSAT, we now discuss the influence of the global constraints (Equations 5 and 6). For this experiment, we generated 40 problem instances comprising 15 nodes and 5 to 40 tasks with a step size of 2. The available memory was randomly chosen from the domain 0 to 6, and the required memory for tasks was randomly chosen from the domain 0 to 4. All runs were carried out 5 times using the same problem instances. Like for all experiments, we set the time boundary for computing solution to 30 seconds[1]. Furthermore, we applied the same problem instances to the ASP program with global constraints and to the other without.

Figure 4 depicts the obtained results. There, we see the runtime in red for the instances with global constraints and for the case without them. In most cases, the solution with global constraints runs faster than the one without them. The dotted line shows the probability that an instance is SAT. Hence,

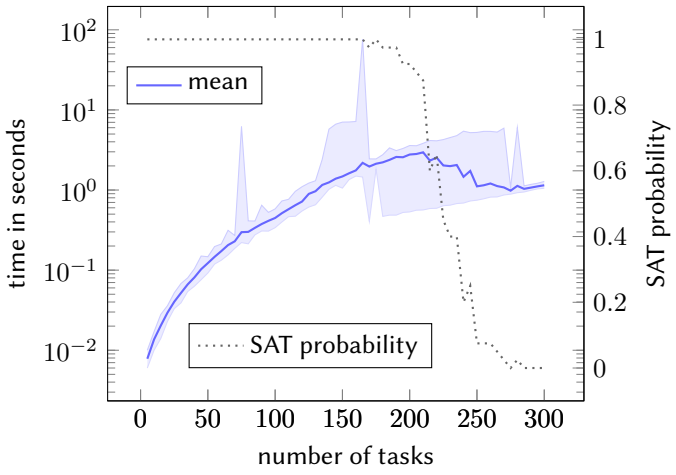
[1]It is worth noting that even when setting the time boundary, in some experiments this was ignored by the ASP solver. Hence, we see larger computing times in the figures.



(a) 25 nodes



(b) 50 nodes



(c) 75 nodes

Figure 3: Mean solving runtime of instance of 25, 50, and 75 nodes and a variable number of tasks ranging from 5 to 120, 200, and 300, respectively. The minimum, maximum and mean runtime in seconds is provided. In addition, the dotted line indicates the probability of satisfiability as a function of the number task.

we see that the runtime increases substantially as the number of UNSAT instances increases, but only when global constraints are not considered. The reason this is that, without global constraints, we need

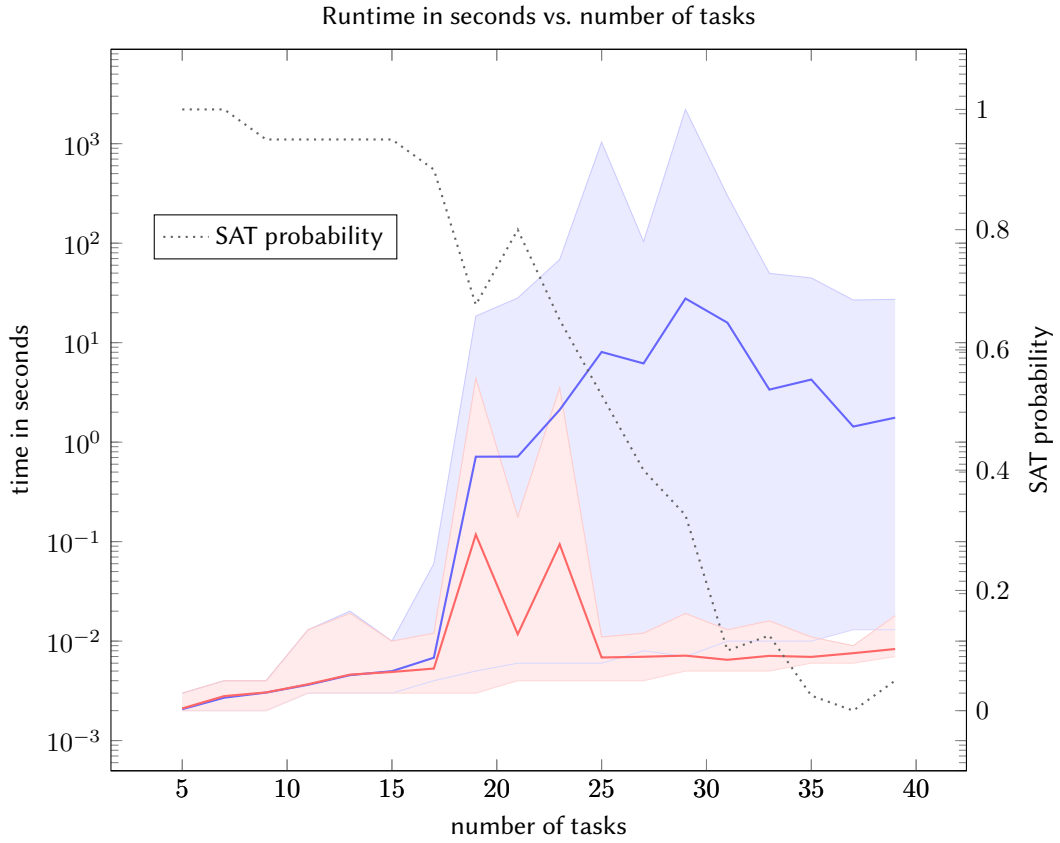


Figure 4: Mean solving runtime of instance of 15 nodes and a variable number of tasks ranging from 5 to 40 with global constraints shown in red and without global constraints given in blue. The areas in light red and light blue indicate the area of runtime between the minimum and maximum when using global constraints and no global constraints respectively.

to consider the entire available search space before being sure there is no solution. This can be easily avoided by introducing global constraints.

What is also interesting is that there are two cases where we see a substantial increase in runtime when using global constraints. A further, more detailed investigation revealed exactly the following two example instances:

- 19 tasks (EXAMPLE_35_8) UNSAT - 34 memory requests in total and 35 memory available.
- 23 tasks (EXAMPLE_18_10) UNSAT - 36 memory requests in total and 37 memory available

In both cases, the global constraints are not violated, yet there is still no solution. There are several possibilities behind such problem instances.

Maximum requested memory of task cannot be satisfied: Let us assume, 3 nodes n_1, n_2, n_3 each providing 3 kilobytes of memory. Furthermore, we have 4 tasks, t_1 to t_4, where all but t_4 require 1 kilobyte of memory, and t_4 requires 4 kilobytes. For such a problem, there is no solution because none of the computing nodes has enough memory. Hence, an additional global constraint of the following form would allow us to terminate the search fast.

$$\forall t \in \mathbf{T} : m(t) \rightarrow \exists n \in \mathbf{N} : m(t) \leq m(n) \tag{8}$$

The computing nodes do not provide enough memory to serve enough tasks: Let us assume, 3 nodes n_1, n_2, n_3 each providing 3 kilobytes of memory. Furthermore, we have 4 tasks, t_1 to

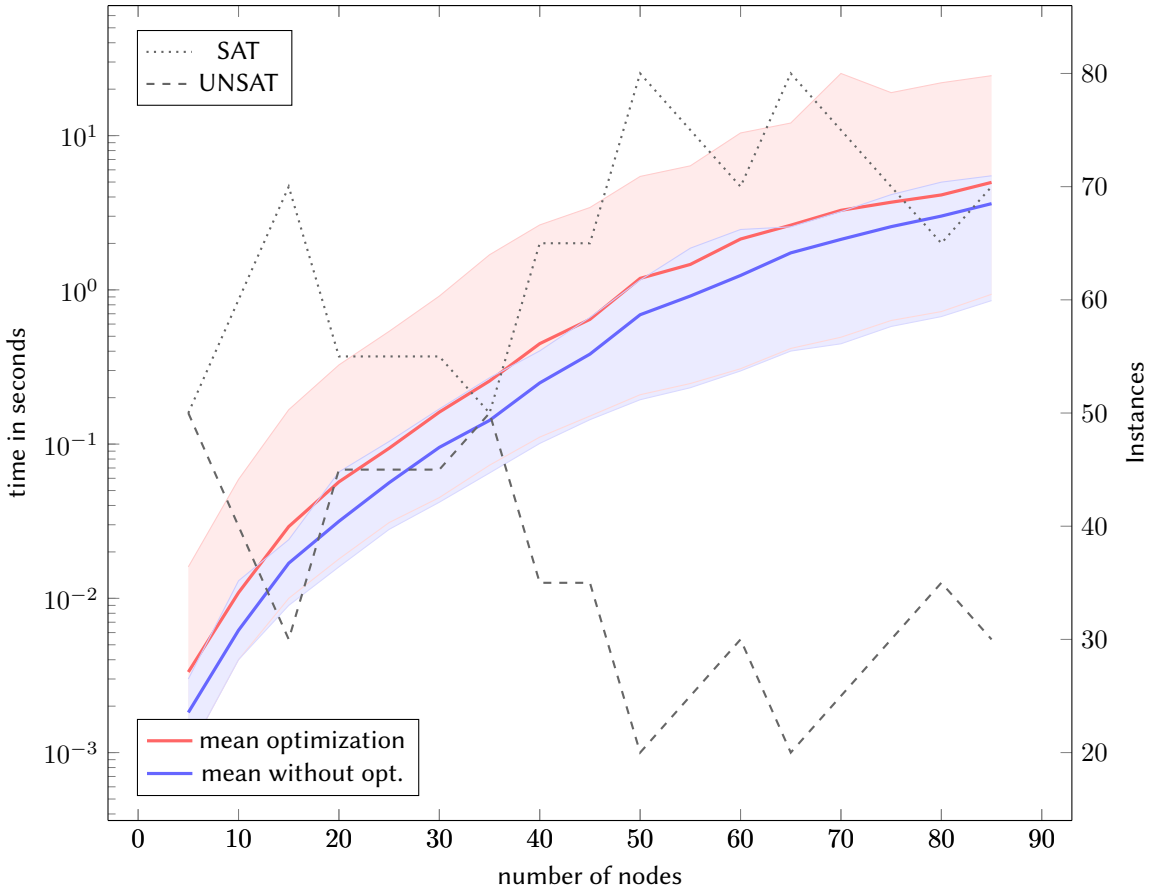


Figure 5: Mean solving runtime for 5 to 85 nodes and a corresponding number of tasks for the optimization problem (in red) compared to the problem solution without optimization (in blue). Besides the minimum, maximum, and mean runtime in seconds, this figure also provides information regarding the number of SAT and UNSAT instances.

t_4, each requiring 2 kilobytes. Obviously, there is no solution, because we can fit only one task to each node, leaving one task without an assignment. The two problem instances mentioned, EXAMPLE_35_8 and EXAMPLE_18_10, fall into this category. It is worth noting that for this category„ formalizing a global constraint seems to be harder.

3.2.4. Optimization problem experiment

For the optimization variant of the task assignment problem, we used the data from our initial experiment. We only set the number of nodes to 85 because larger instances could not be solved within the given time budget. Hence, we considered only SAT and UNSAT instances.

From Figure 5, we see that the computation of the optimized solution follows the same curve as the non-optimized variant but takes longer. Hence, when searching for an optimal solution, we need to consider a much higher maximum runtime. Interestingly, the optimized case's runtime is almost identical to that of the unoptimized case. The reason for this observation might be that the experiments also capture UNSAT instances, which do not require additional search for optimality. Only for SAT instances, we see an increase in runtime.

4. Discussions

The experimental evaluation comprises, in sum, 6,280 instances of the task assignment problem. We generated the instances randomly using a program. Hence, not all the instances might be different.

However, the experimental evaluation is extensive and reveals certain behavioral specificities of the ASP solution of the task assignment problem worth further discussion and summarization.

- First, there is a substantial bandwidth between the minimum and maximum runtime, especially for larger problem instances. However, it is possible to find a probability distribution of the runtime for any number of nodes and tasks, which can serve as a basis for calculating a runtime guarantee for practical applications. For example, let us have a look at Figure 6, which depicts a cumulative histogram of 8,100 runs of example instances with 50 nodes. At about 1 second, we reach 8,000 runs. Hence, with a probability of 0.9877, the runtime for problem instances comprising 50 nodes and a varying number of tasks is less than or equal to 1 second.
- Second, there is also a substantial difference between the time required for SAT and UNSAT instances. For SAT instances, the runtime increases when the number of solutions is limited, requiring consideration of the entire search space. Most UNSAT instances can be found using global constraints. For instances where the currently used global constraints are not violated but the problem instance is UNSAT, we also experience extended runtime. It would be interesting to further investigate this topic to elaborate on the different reasons behind this observation.
- Similarly to the previous findings, the runtime varies most in the transition phase between SAT and UNSAT instances. These are the hardest instances to solve. Before the transition phase, the runtime varies but – with exceptions – not substantially. After the transition phase, UNSAT seems to be easily ensured by the global constraints.
- The global constraints have a huge impact on runtime performance, especially when the problem size increases. Without global constraints, the ASP-based solution to the problem is hardly feasible. Hence, using global constraints is essential, and identifying additional ones is worth investigating.
- Optimization increases runtime and should be considered when required for applications in practice. We did not use any optimization for running the ASP solver *clingo*. The use of multi-cores might be an advantage when searching for an optimized solution, and it is left for future research.
- We identified several UNSAT instances in which the global constraints are satisfied. Hence, it is worth considering additional global constraints for further improving performance. When searching for methods to reduce the search space, such as symmetry breaking [20, 21], might be appropriate. This might reduce search, especially in optimization.

The conducted experimental evaluation allows us to discuss the practicability of implementing the task assignment problem using ASP. Besides the advantage of providing declarative knowledge for representing the problem, we can also estimate the runtime based on the results. From Figure 2, we see that for instances having at most 20 nodes, we obtain results in less than 0.1 seconds. Of course, this result does not hold in general and depends on the available computational resources. Experiments might be re-executed on other hardware and software, where this paper serves as a blueprint.

There are also several threats to the validity of the experimental evaluation that warrant mention. Such threats include the design of the experiments, the generation of the problem instances, and their analysis. We generated and executed all problem instances automatically using Java programs. These programs have been tested and run several times to ensure they are functioning correctly. Problem instances were generated using specific domains for the parameters, with concrete values selected at random. Hence, the generated instances show a wide variety, but random selection might lead to the omission of certain, specifically harder instances, which is hardly avoidable. The evaluation is again based on a program that was tested on smaller examples. Hence, it is unlikely that any severe fault was undetected. There is, of course, an influence from the hardware and software used to carry out the experiments. Because all experiments were carried out on the same hardware and software, we do not expect any significant influence on the comparison of results. We do not expect that changes in hardware or software will substantially affect the outcomes of the experimental evaluation and comparison. Only the absolute runtime values might vary.

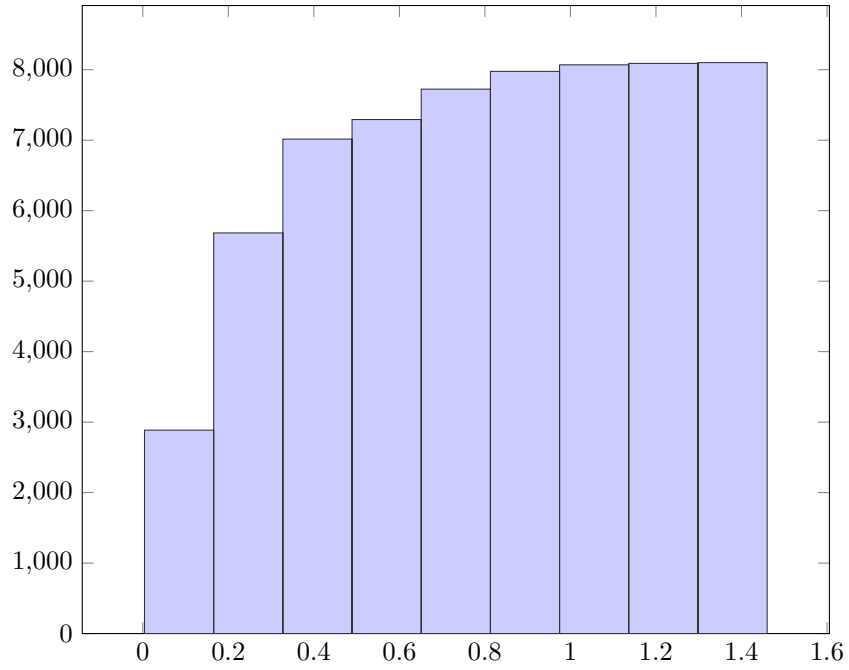


Figure 6: The cumulative histogram of the runtime in seconds of all 5 runs and 1,620 problem instances comprising 50 nodes, which sums up to 8,100 runs. We see, for example, that almost 3,000 instances require less than 0.2 seconds, and about 5,900 instances need less than 0.4 seconds, etc. for computing a solution.

5. Conclusions

This paper presents an exhaustive experimental evaluation of the ASP solution to the task assignment problem. The evaluation is based on 6,280 randomly generated problem instances, covering examples up to 150 computing nodes. The experimental evaluation identified the transition phase between satisfiable and unsatisfiable problem instances and the importance of global constraints for identifying unsatisfiable instances as early as possible. For the latter, we discussed two reasons for the need to develop a corresponding global constraint. Furthermore, we present the first experiments in which the task was to compute an optimal solution, i.e., one with the fewest required computing nodes.

Future research should address adding symmetry-breaking mechanisms and a more detailed analysis of the quality of computed solutions, i.e., their closeness to optimality. Furthermore, it is necessary to compare the ASP solution with other algorithmic approaches, given the examples provided in the study. Finally, we want to carry out research on the effects of *clingo*'s parameters on its runtime behavior.

Acknowledgments

The work was supported by the Austrian Science Fund (FWF) Cluster of Excellence Bilateral AI under contract number 10.55776/COE12.

Declaration on Generative AI

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

References

[1] M. Nica, B. Peischl, F. Wotawa, A constraint model for automated deployment of automotive control software, in: Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008, Knowledge Systems Institute Graduate School, 2008, pp. 899–904.

[2] R. Rotaeche, A. Ballesteros, J. Proenza, Speeding task allocation search for reconfigurations in adaptive distributed embedded systems using deep reinforcement learning, Sensors 23 (2023) 548. URL: https://doi.org/10.3390/s23010548. doi:10.3390/S23010548.

[3] A. Ballesteros, M. Barranco, J. Proenza, L. Almeida, F. Pozo, P. Palmer-Rodríguez, An infrastructure for enabling dynamic fault tolerance in highly-reliable adaptive distributed embedded systems based on switched ethernet, Sensors 22 (2022) 7099. URL: https://doi.org/10.3390/s22187099. doi:10.3390/S22187099.

[4] P. Ferraris, V. Lifschitz, Mathematical foundations of answer set programming, in: We Will Show Them! (1), College Publications, 2005, pp. 615–664.

[5] T. Eiter, G. Ianni, T. Krennwallner, Answer set programming: A primer, in: S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.-C. Rousset, R. A. Schmidt (Eds.), Reasoning Web. Semantic Technologies for Information Systems: 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 40–110. URL: https://doi.org/10.1007/978-3-642-03754-2_2. doi:10.1007/978-3-642-03754-2_2.

[6] F. Wotawa, Using answer set programming for assigning tasks to computing nodes, in: É. Vareilles, C. Grosso, J. M. Horcas, A. Felfernig (Eds.), Proceedings of the 26th International Workshop on Configuration (ConfWS 2024) co-located with the 30th International Conference on Principles and Practice of Constraint Programming (CP 2024), Girona, Spain, September 2-3, 2024, volume 3812 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 64–67. URL: https://ceur-ws.org/Vol-3812/paper9.pdf.

[7] F. Wotawa, J. Proenza, M. Barranco, A. Ballesteros, The task assignment problem for safety-critical networks considering communication and criticality, in: C. Grosso, E. Sandrin, V. Le (Eds.), Proceedings of the 27th International Workshop on Configuration (ConfWS 2025) co-located with the 28th European Conference on Artificial Intelligence (ECAI 2025), Bologna, Italy, October 25-26, 2025, CEUR Workshop Proceedings, CEUR-WS.org, 2025, pp. 30–37. URL: https://ceur-ws.org/Vol-4149/paper3.pdf.

[8] M. Abseher, M. Gebser, N. Musliu, T. Schaub, S. Woltran, Shift design with answer set programming, in: F. Calimeri, G. Ianni, M. Truszczynski (Eds.), Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings, volume 9345 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 32–39. URL: https://doi.org/10.1007/978-3-319-23264-5_4. doi:10.1007/978-3-319-23264-5_4.

[9] S. Mishra, Product configuration in answer set programming, Electronic Proceedings in Theoretical Computer Science 345 (2021) 296–304. URL: http://dx.doi.org/10.4204/EPTCS.345.46. doi:10.4204/eptcs.345.46.

[10] R. Comploi-Taupe, G. Friedrich, T. Niestroj, Dynamic aggregates in expressive ASP heuristics for configuration problems, in: J. M. Horcas, J. A. Galindo, R. Comploi-Taupe, L. Fuentes (Eds.), Proceedings of the 25th International Workshop on Configuration (ConfWS 2023), Málaga, Spain, September 6-7, 2023, volume 3509 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 75–84. URL: https://ceur-ws.org/Vol-3509/paper11.pdf.

[11] N. Rühling, T. Schaub, T. Stolzmann, Towards a formalization of configuration problems for asp-based reasoning: Preliminary report, in: J. M. Horcas, J. A. Galindo, R. Comploi-Taupe, L. Fuentes (Eds.), Proceedings of the 25th International Workshop on Configuration (ConfWS 2023), Málaga, Spain, September 6-7, 2023, volume 3509 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 85–94. URL: https://ceur-ws.org/Vol-3509/paper12.pdf.

[12] V. Cacchiani, M. Iori, A. Locatelli, S. Martello, Knapsack problems – an overview of recent advances.

part i: Single knapsack problems, Computers & Operations Research 143 (2022) 105692. URL: https://www.sciencedirect.com/science/article/pii/S0305054821003877. doi:https://doi.org/10.1016/j.cor.2021.105692.

[13] V. Cacchiani, M. Iori, A. Locatelli, S. Martello, Knapsack problems – an overview of recent advances. part ii: Multiple, multidimensional, and quadratic knapsack problems, Computers & Operations Research 143 (2022) 105693. URL: https://www.sciencedirect.com/science/article/pii/S0305054821003889. doi:https://doi.org/10.1016/j.cor.2021.105693.

[14] B. Korte, J. Vygen, Bin-packing, in: Combinatorial Optimization: Theory and Algorithms, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 426–441. URL: https://doi.org/10.1007/3-540-29297-7_18. doi:10.1007/3-540-29297-7_18.

[15] E. G. Coffman, M. R. Garey, D. S. Johnson, Approximation Algorithms for Bin-Packing — An Updated Survey, Springer Vienna, Vienna, 1984, pp. 49–106. URL: https://doi.org/10.1007/978-3-7091-4338-4_3. doi:10.1007/978-3-7091-4338-4_3.

[16] P. Cheeseman, B. Kanefsky, W. M. Taylor, Where the really hard problems are, in: Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI'91, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991, p. 331–337.

[17] D. Mitchell, B. Selman, H. Levesque, Hard and easy distributions of sat problems, in: Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI'92, AAAI Press, 1992, p. 459–465.

[18] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Clingo = ASP + control: Preliminary report, CoRR abs/1405.3694 (2014).

[19] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot asp solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[20] B. Benhamou, Dynamic and static symmetry breaking in answer set programming, in: K. McMillan, A. Middeldorp, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 112–126.

[21] J. Devriendt, B. Bogaerts, Breakid: Static symmetry breaking for asp (system description), 2016. URL: https://arxiv.org/abs/1608.08447. arXiv:1608.08447.

Use Cases for Generative AI in Product Configuration

Konstantin Herud[1], Joachim Baumeister[1,2]

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

Abstract

Generative AI can support knowledge engineering (KE) by helping experts work across informal evidence, formal models, and natural-language interaction. This paper defines a research agenda for such support in industrial KE. We focus here on product configuration, using it as a concrete industrial KE setting, while treating the proposed collaboration modes as broadly applicable to KE tasks that involve actionable knowledge artifacts. Generative AI may help derive candidate knowledge base changes from heterogeneous evidence, support impact analysis within an existing knowledge base, and debug observed outcomes such as rejected configurations or failing regression tests. Across these cases, the central problem is grounded and reviewable collaboration between human experts and generative AI. The paper argues that impact analysis and debugging offer near-term paths for method development, while acquisition from heterogeneous evidence is the harder long-term test for generative AI in knowledge engineering.

Keywords

Knowledge Engineering, Generative AI, Product Configuration

1. Introduction

Generative AI is changing software engineering practice. Engineers use it to draft code and tests, explore unfamiliar code bases, summarize alternatives, and shorten iteration cycles. Early reports and empirical studies suggest that these uses can yield real productivity gains, even if the gains vary by task and setting [1, 2, 3, 4]. That shift matters beyond software engineering.

Knowledge engineering (KE) is likely to undergo a similar shift. In many KE domains, a knowledge base can determine which system behaviors are permitted, how decisions are made, and how organizational processes respond to change [5, 6]. In industrial product configuration, for example, knowledge bases affect which product variants are accepted, how conflicts are explained, and how later modifications propagate through a configurator [7, 8, 9, 10]. Small semantic errors can therefore have consequences that extend well beyond drafting quality.

Generative AI also changes how experts can interact with knowledge artifacts. It enables forms of interaction that were previously difficult or too costly to support at useful levels of flexibility. A model can help turn heterogeneous evidence into candidate knowledge base changes, answer natural-language questions about a large and evolving knowledge base, and translate low-level formal evidence into explanations that non-experts can inspect and act on [11, 12]. These capabilities are promising precisely because KE rarely starts from a clean specification and rarely ends with a single act of rule authoring. Figure 1 illustrates this in a KE authoring environment: the expert still navigates and edits the knowledge base, while the assistant is embedded next to the artifact it must inspect, explain, and eventually modify.

Yet KE currently lacks principled methods for using these capabilities well. Recent empirical work with knowledge engineers points to open issues around prompting skills, evaluation of generated outputs, trust, and responsible use [13]. The central challenge is how humans and generative AI should collaborate when the result must remain grounded in source evidence, reviewable by experts, behaviorally reliable, and subject to human responsibility for acceptance and interpretation [14, 15]. This position paper focuses on product configuration because it makes these requirements especially

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany
konstantin.herud@denkbares.com (K. Herud); joba@uni-wuerzburg.de (J. Baumeister)
© 2026 This work is licensed under a "CC BY 4.0" license.

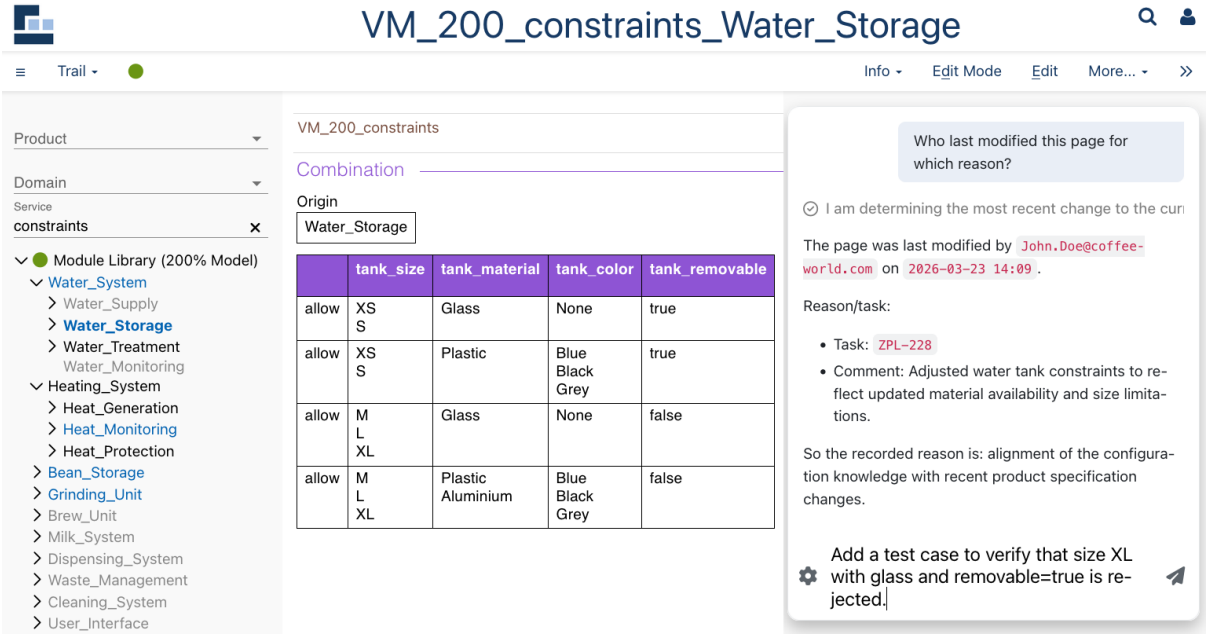


Figure 1: Example knowledge authoring environment with module navigation, an editable knowledge base fragment, and an integrated generative AI assistant, illustrating how such support can become part of existing KE tools.

concrete, but the proposed use cases are intended as instances of a broader agenda for generative AI in knowledge engineering.

Section 2 characterizes KE in industrial manufacturing and uses product configuration as a running example. Section 3 develops three use cases that expose concrete research questions: acquisition from heterogeneous evidence, knowledge base research and impact analysis, and debugging of observed behavior. Section 4 returns to the broader implication and argues for a sustained methods agenda for generative AI in KE.

2. Knowledge Engineering in Industrial Manufacturing

We focus on industrial manufacturing because several hard properties of knowledge engineering appear there at once. Knowledge has to be assembled from heterogeneous evidence, negotiated across organizational roles, encoded in artifacts that directly affect system behavior, and maintained under continual change [5, 6].

Product configuration for a coffee-machine manufacturer provides a concrete running example. In our example, the company sells product families across markets, installation contexts, and feature packages, and it relies on a knowledge base to decide which combinations are admissible, how conflicts are explained, and how changes can be introduced without breaking existing behavior [16, 10].

Throughout the paper, the operational knowledge base is represented using a domain-specific language (DSL); here the language COOM [17]. This assumption reflects many industrial KE environments, where a DSL can make domain concepts explicit, support modular authoring, enable static checks and regression tests, and provide stable references for explanations and impact analysis [18, 19]. These properties are also attractive for agentic AI systems, because they give the model a text-based artifact to inspect, edit, test, and cite. The use cases, however, do not depend on DSLs specifically. Other knowledge representations, such as rule-based models, ontologies, logic programs, or hybrid representations, could support similar collaboration patterns if they expose inspectable structure [20, 5].

2.1. Heterogeneous Knowledge Sources

In an industrial setting, relevant knowledge is rarely located in one clean specification or one formal model. A knowledge engineer working on the coffee-machine configurator may need to consult installation manuals in PDF form, multi-sheet option spreadsheets, enterprise resource planning (ERP) and product lifecycle management (PLM) data, supplier documents, certification records, service reports, issue trackers, and the current knowledge base itself. These sources differ in structure, terminology, granularity, update rhythm, and authority. Some are curated and stable, some are provisional, and some reflect operational practice rather than official policy [21, 6].

The harder problem is that intent is often implicit. Constraints are encoded in tables, footnotes, examples, naming conventions, and historically evolved modeling choices. The same concept may appear in several places with different wording, different scope, or a different revision status. A market-specific power requirement for a coffee machine, for example, may be stated in a manual, reflected in a supplier data sheet, partly encoded in ERP material variants, and only indirectly represented in existing rules. Knowledge engineering therefore begins with evidence triage and interpretation. Formalization comes later, after the engineer has determined which sources matter, how they relate, and what they actually commit the organization to [21, 5].

2.2. Stakeholder Diversity

Industrial knowledge is also distributed across stakeholders who each contribute a different part of the product reality. In the coffee-machine example, engineering defines technical dependencies, sales frames sellable options and market packages, compliance tracks certification and regional obligations, manufacturing cares about buildability, and knowledge engineering turns these inputs into a maintained operational model. These use different vocabularies, ask different questions, and care about different consequences. Conflicts are therefore normal. One source may describe an option as generally available while another restricts it to a subset of machine variants. Engineering may group components by internal architecture while sales uses product-line language that cuts across those boundaries. Compliance may insist on a regional distinction that other roles have previously ignored. Knowledge engineering has to reconcile these differences, set priorities, and make modeling conventions explicit.

2.3. Continual Change

The artifacts produced by knowledge engineering are actionable. In a coffee-machine configurator, a small semantic error can block valid orders, admit invalid combinations, or distort the explanation given to sales, service, or customers. The artifacts also evolve continuously. New product variants are introduced, region-specific requirements change, supplier portfolios shift, and certification constraints are revised. KE spans recurring activities rather than a one-time modeling step: introducing changes, understanding the current knowledge base, estimating the impact of planned modifications, and debugging observed outcomes [10, 22]. Generative AI may help across all of these activities, but it may also increase the number of plausible candidate changes and the burden of review. Once KE is understood as work on an evolving ecosystem of artifacts rather than isolated rule authoring, the need for principled methods of human-generative-AI collaboration becomes concrete.

3. Use Cases for Generative AI in Knowledge Engineering

Generative AI is useful in this setting because it can operate across representation boundaries. In one interaction loop it can move between source documents, DSL modules, tests, traces, and formal evidence while still supporting natural-language exchange with the human expert [23, 24]. More generally, the same pattern applies whenever the knowledge representation exposes structure that can be retrieved, checked, and linked back to user-facing explanations. We therefore concentrate on three recurring collaboration modes that arise naturally in industrial KE work: from evidence to change, from question

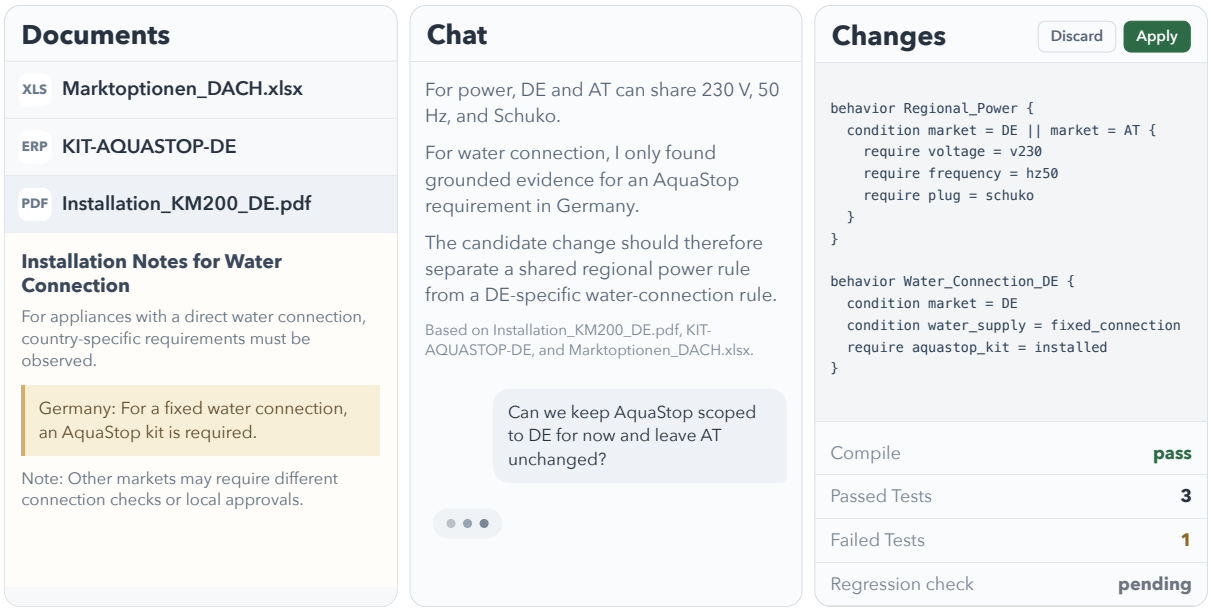


Figure 2: Conceptual workspace for evidence-to-change collaboration. The left panel shows heterogeneous source material, the middle panel supports clarification, and the right panel presents a candidate DSL patch with basic compilation and test feedback. Users primarily interact with the middle panel. Answers directly link back to source documents.

to impact understanding, and from observed behavior to debugging. These use cases make the methods agenda visible as concrete research fronts.

3.1. Acquisition from Heterogeneous Evidence

The first collaboration mode starts with a recurring KE task: turning heterogeneous evidence into candidate operational knowledge. Consider a knowledge engineer who has to implement a market update for a coffee-machine family. The available material includes an installation manual, a multi-sheet market-options spreadsheet, ERP material data, and the current knowledge base. Relevant fragments are concrete and unevenly distributed across these sources. The manual states `230 V / 50 Hz / Schuko-Stecker` (Schuko plug). Another source specifies that machines with `Festwasseranschluss` (permanent water connection) require an `AquaStop-Kit`. ERP data may refer to a material such as `KIT-AQUASTOP-DE`. The task is not complete once these fragments have been found. The engineer has to produce a knowledge change that can be reviewed, tested, and accepted in the configurator.

A plausible collaboration method would let the human provide the gathered sources and the current knowledge base, then let the agent extract relevant facts and relations, connect them to the existing vocabulary and likely points of integration, ask focused clarification questions where scope remains unclear, and draft a candidate DSL change together with checks. Figure 2 sketches such a workspace. The left panel organizes source material, the middle panel supports the interaction and clarification loop, and the right panel turns the result into an inspectable patch with basic feedback such as compilation and regression status. The human remains responsible for resolving ambiguity and deciding whether the proposed change is acceptable.

> **Research Question 1** Does the generative AI shorten the path from heterogeneous industrial evidence to an accepted knowledge change while preserving grounding, behavioral correctness, and reviewability?

The process begins with documentation understanding. Source documents are large, heterogeneous, and difficult to interpret reliably. The method has to identify which sections, tables, notes, and relations matter for the current task and which do not. In industrial documents, relevant intent is often expressed

indirectly through examples, footnotes, naming conventions, or exceptions. The source view in Figure 2 illustrates this first step.

Once relevant fragments have been identified, they have to be related to the existing knowledge base. Extracted industrial language has to be aligned with the current formal vocabulary, the existing rule structure, and the likely points of integration in the knowledge base. A phrase such as `230 V / 50 Hz / Schuko-Stecker` is useful only if it can be related to existing concepts for market, voltage, frequency, and plug type, and only if the method can determine whether similar behavior is already encoded elsewhere. The right panel in Figure 2 represents this alignment problem. The system has to draft a change that fits the current artifact rather than producing an isolated rule fragment.

Even after extraction and alignment, ambiguity remains. The available sources may support an `AquaStop-Kit` requirement while leaving open whether it applies only in Germany, across the full Germany-Austria-Switzerland market, or only to a subset of machine families. A collaboration method therefore needs a disciplined clarification step. It should ask targeted questions tied to concrete evidence and explicit uncertainty, rather than replacing missing information with fluent guesses [14, 25]. The chat workspace in Figure 2 illustrates this middle step.

3.2. Research within the Knowledge Base

The second collaboration mode begins inside an existing knowledge base. In the coffee-machine configurator, different actors query the same artifact in different vocabularies. Sales may ask "Which machines qualify for office installation?". Engineering may ask "Is TB-12 approved on the compact hydraulics?". A knowledge engineer may ask "If MilkSystem is decomposed, what changes?". The first two questions show that the knowledge base may serve as the company's practical ground truth for several roles.

Suppose the current concept `MilkSystem` has grown into a mixed abstraction that bundles milk source choices with cleaning obligations. A knowledge engineer considers decomposing it into a cleaner internal structure and needs to understand, before editing anything, which product contexts, constraints, explanations, and tests would be affected.

A plausible collaboration method would treat this as an impact-analysis session. The human asks a short, partly underspecified question. The system interprets the intended concept, distinguishes reusable library knowledge from product-specific usage, consults DSL modules, tests, explanation templates, and dependency information, and returns a natural-language impact brief. Figure 3 sketches such a session. The left panel shows the question and answer, while the right panel grounds the answer in structural evidence from the knowledge base. The human remains responsible for judging whether the analysis is sufficient and for deciding on the actual change.

> **Research Question 2** How can generative AI support understanding and impact analysis in complex knowledge bases?

The interaction starts from ambiguous, role-specific questions. They are often short, underspecified, and phrased in stakeholder-specific language. The system has to resolve terminology, infer missing scope only where justified, and surface ambiguity when clarification is needed. The left panel in Figure 3 illustrates this problem. A compact question has to be expanded into an answer that matches the intended modeling concern rather than a nearby concept.

Even when the intended concept is clear, reconstructing it across layered and modular artifacts remains difficult. The relevant answer may span reusable library knowledge, product-specific knowledge, tests, and compiled or derived artifacts. A useful system has to preserve these context boundaries instead of flattening them into one undifferentiated summary. The right panel in Figure 3 represents this layered context. The method has to distinguish what belongs to general library structure, what belongs to specific coffee-machine families, and what belongs to derived artifacts that constrain behavior or explanation.



Figure 3: Conceptual workspace for impact analysis within an existing knowledge base. The left panel turns a short question into a natural-language impact brief, and the right panel grounds that brief in structural evidence.

Impact analysis then has to be grounded in executable structure. Impact can often not be answered from text or name matching alone. It requires structural grounding through symbolic references, dependency information, or similar relationships that expose how behavior is connected [22]. Reviewability cuts across all three challenges. The answer has to expose assumptions, uncertainty, and evidence in a form that humans can inspect.

3.3. Debugging of Outcomes and Regressions

The third collaboration mode begins at a later point in the lifecycle. Here the knowledge base has already produced an observable outcome, and that outcome now has to be debugged. Consider an office coffee-machine configuration with `MilkSystem = FreshMilk`. The configurator rejects it. The environment can already provide part of the formal basis of that rejection, for example the selected values, the relevant DSL fragments, and a minimal conflict explanation [26, 27, 28]. Yet these artifacts are still hard to interpret in context. What the human needs is a grounded debugging explanation that shows why the configuration is rejected and what broader modeling issue the conflict points to. Regression tests are a related case: a failing test also creates an observed outcome that has to be debugged rather than merely reported.

A plausible collaboration method would begin from a selected observed case. The environment supplies the relevant formal evidence, such as configuration values and violated constraints. The agent then produces a contextual debugging explanation in human language. It should make clear what is formally established by the artifacts, how those artifacts fit together causally, and what higher-level interpretation they support. Figure 4 sketches such a workspace. The right panel records the rejected case together with the supporting rule fragments, and the left panel turns that material into a debugging explanation that a human can inspect, question, and refine. The human remains responsible for deciding whether the explanation is adequate and what modeling action should follow.

> **Research Question 3** How can generative AI support grounded debugging of observed behavior in complex knowledge bases?

Debugging starts from complex formal context. The formal basis of an observed outcome may already be available, but making sense of it is still non-trivial. Even a small case can require following multi-step dependencies, respecting modular boundaries, and understanding reused knowledge in its specific

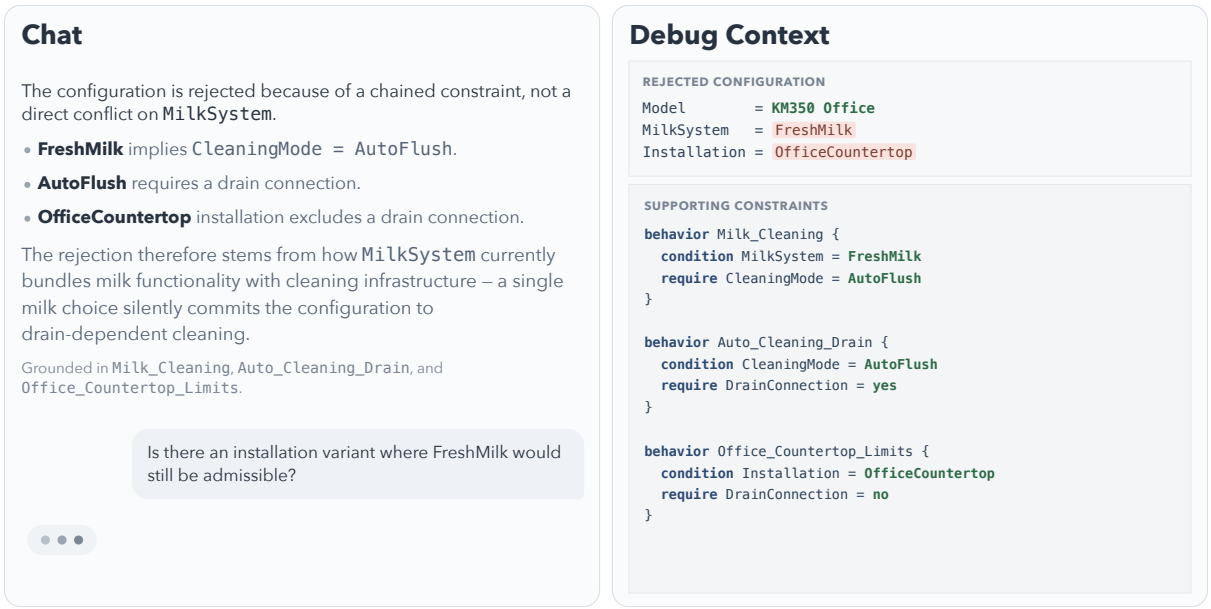


Figure 4: Conceptual workspace for debugging observed behavior. The right panel shows the rejected case together with relevant knowledge base fragments, and the left panel turns that formal material into a contextual debugging explanation and follow-up interaction.

product context [26, 28]. The left panel in Figure 4 illustrates this problem. A generative system can easily produce a locally plausible explanation that ignores the actual dependency chain or introduces causal claims not supported by the provided artifacts [25, 29].

From there, the method has to move toward faithful abstraction across representational levels. A useful debugging explanation should not simply restate raw constraints, but it also cannot collapse into a fluent story that hides scope conditions, uncertainty, or model-specific meaning. The right panel in Figure 4 illustrates the required move from formal evidence to stakeholder-usable language. The central problem is controlled abstraction: the method has to contextualize the formal basis while preserving traceability to it and exposing what is interpretation rather than formal fact.

4. Conclusion

The three use cases expose different levels of difficulty and different near-term opportunities. Acquisition from heterogeneous evidence is the hardest case by a wide margin. It combines document understanding, source authority, terminology alignment, ambiguity resolution, integration into an existing knowledge base, and the generation of a candidate change whose acceptance would alter system behavior. Its success criterion is correspondingly strong: the output is a candidate knowledge base change that could be reviewed and accepted, not merely an analysis that informs later work. Research within an existing knowledge base is more tractable in the near term because it starts from a grounded artifact, can rely on structural information already present in the environment, and aims at an impact brief rather than an immediate change. Debugging observed behavior is promising in a different way. When traces, violated tests, unsatisfiable cores, or linked DSL rules are available, the formal basis is often stronger than in the other two cases. Its open difficulty is faithful abstraction: it is still unclear how far formal evidence can be translated into stakeholder language without losing too much of the causally relevant structure.

Current generative AI changes the space of possible KE interactions because one natural-language interaction layer can now mediate among documents, knowledge base modules, tests, and formal evidence [11, 12]. The consequence is flexibility as much as speed. Knowledge engineers can query, compare, summarize, and reformulate artifacts across representation boundaries that previously required separate tools or bespoke interfaces. Current models are already highly capable in areas such as programming, but it remains unclear how far those capabilities transfer to private, niche, organization-

specific KE domains such as industrial product configuration [1]. Performance in KE will likely depend strongly on the harness around the model: how it accesses source documents, knowledge base structure, tests, traces, and explicit signals of ambiguity. Hallucination remains unresolved [25]. In KE, fluent output is not an adequate success criterion. The relevant question is under which collaboration structures current models become dependable enough to support real knowledge work.

Near-term advances in context handling, multimodal document processing, tool use, and tighter integration with structured artifacts will expand what is feasible in all three cases. In acquisition workflows, larger context capacity may reduce the need for manually designed intermediate document representations in some settings. That does not remove the core problem, which is still to move from heterogeneous evidence to an accepted and behaviorally correct knowledge change. More generally, stronger models widen the feasible interaction space, but they do not answer the central KE questions. As capabilities improve, the bottleneck shifts toward method design: how work is divided between humans and models, how outputs are grounded in sources and formal artifacts, how ambiguity is surfaced and resolved, how review is supported, and how trust is calibrated without over-relying on fluent output [14, 15, 25, 13]. This is why the three use cases matter as a research progression rather than as isolated examples. The more grounded cases, especially impact analysis and parts of debugging, can establish methods for harness design, grounding, and review. The harder acquisition case then becomes the longer-term test of whether those methods scale to messy industrial evidence and accepted change. The next step for KE is to turn emerging capability into collaboration methods that stay grounded, reviewable, and fit for industrial knowledge work.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT and OpenAI Codex, to support drafting, editing, and LaTeX troubleshooting. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Chen, et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).

[2] P. Vaithilingam, T. Zhang, E. L. Glassman, Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models, in: Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, 2022, pp. 1–7. doi:10.1145/3491101.3519665.

[3] S. Barke, M. B. James, N. Polikarpova, Grounded Copilot: How programmers interact with code-generating models, Proceedings of the ACM on Programming Languages 7 (2023) 85–111. doi:10.1145/3586030.

[4] S. Peng, E. Kalliamvakou, P. Cihon, M. Demirer, The impact of AI on developer productivity: Evidence from GitHub Copilot, arXiv preprint arXiv:2302.06590 (2023).

[5] R. Studer, V. R. Benjamins, D. Fensel, Knowledge engineering: Principles and methods, Data & Knowledge Engineering 25 (1998) 161–197. doi:10.1016/S0169-023X(97)00056-6.

[6] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. Van de Velde, B. Wielinga, Knowledge Engineering and Management: The CommonKADS Methodology, MIT Press, Cambridge, MA, 1999.

[7] S. Mittal, F. Frayman, Towards a generic model of configuration tasks, in: Proceedings of the 11th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1989, pp. 1395–1401.

[8] D. Sabin, R. Weigel, Product configuration frameworks—a survey, IEEE Intelligent Systems 13 (1998) 42–49. doi:10.1109/5254.708432.

[9] M. Stumptner, An overview of knowledge-based configuration, AI Communications 10 (1997) 111–125.

[10] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen, Knowledge-Based Configuration: From Research to Business Cases, Morgan Kaufmann Publishers, Waltham, MA, USA, 2014. doi:10.1016/B978-0-12-415817-7.00029-3.

[11] B. P. Allen, L. Stork, P. Groth, Knowledge engineering using large language models, Transactions on Graph Data and Knowledge 1 (2023) 3:1–3:19. doi:10.4230/TGDK.1.1.3.

[12] R. Buchmann, J. Eder, H.-G. Fill, U. Frank, D. Karagiannis, E. Laurenzi, J. Mylopoulos, D. Plexousakis, M. Y. Santos, Large language models: Expectations for semantics-driven systems engineering, Data & Knowledge Engineering 152 (2024) 102324. doi:10.1016/j.datak.2024.102324.

[13] E. Koutsiana, J. Walker, M. Nwachukwu, B. Zhang, A. Meroño-Peñuela, E. Simperl, Knowledge prompting: How knowledge engineers use generative AI, Journal of Web Semantics 88 (2026) 100873. doi:10.1016/j.websem.2025.100873.

[14] S. Amershi, D. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. Iqbal, P. N. Bennett, K. Inkpen, J. Teevan, R. Kikin-Gil, E. Horvitz, Guidelines for human-AI interaction, in: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, ACM, 2019, pp. 1–13. doi:10.1145/3290605.3300233.

[15] J. D. Lee, K. A. See, Trust in automation: Designing for appropriate reliance, Human Factors 46 (2004) 50–80. doi:10.1518/hfes.46.1.50_30392.

[16] T. Soininen, J. Tiihonen, T. Männistö, R. Sulonen, Towards a general ontology of configuration, AI EDAM 12 (1998) 357–372. doi:10.1017/S0890060498124083.

[17] J. Baumeister, K. Herud, J. Reutelshöfer, V. Belli, coom-lang, https://www.coom-lang.org, 2025. Release 06.2025.

[18] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: An annotated bibliography, ACM SIGPLAN Notices 35 (2000) 26–36. doi:10.1145/352029.352035.

[19] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (2005) 316–344. doi:10.1145/1118890.1118892.

[20] T. R. Gruber, A translation approach to portable ontology specifications, Knowledge Acquisition 5 (1993) 199–220. doi:10.1006/knac.1993.1008.

[21] J. H. Boose, B. R. Gaines, Knowledge acquisition for knowledge-based systems: Notes on the state-of-the-art, Machine Learning 4 (1989) 377–394. doi:10.1023/A:1022662924615.

[22] S. A. Bohner, R. S. Arnold (Eds.), Software Change Impact Analysis, IEEE Computer Society Press, Los Alamitos, CA, 1996.

[23] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, D. Kiela, Retrieval-augmented generation for knowledge-intensive NLP tasks, in: Advances in Neural Information Processing Systems, volume 33, 2020, pp. 9459–9474.

[24] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, ReAct: Synergizing reasoning and acting in language models, arXiv preprint arXiv:2210.03629 (2023).

[25] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, P. Fung, Survey of hallucination in natural language generation, ACM Computing Surveys 55 (2023) 1–38. doi:10.1145/3571730.

[26] R. Reiter, A theory of diagnosis from first principles, Artificial Intelligence 32 (1987) 57–95. doi:10.1016/0004-3702(87)90062-2.

[27] U. Junker, QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems, in: Proceedings of the Nineteenth National Conference on Artificial Intelligence, AAAI Press, 2004, pp. 167–172.

[28] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Consistency-based diagnosis of configuration knowledge bases, Artificial Intelligence 152 (2004) 213–234. doi:10.1016/S0004-3702(03)00117-6.

[29] J. Maynez, S. Narayan, B. Bohnet, R. McDonald, On faithfulness and factuality in abstractive summarization, in: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 1906–1919. doi:10.18653/v1/2020.acl-main.173.

The ShadowMaster Problem: A Real-World Case Study in Product Configuration

Joachim Baumeister[1,2], Konstantin Herud[1], Jan Heuer[3,4], Jochen Reutelshöfer[1], Nicolas Rühling[3,4], Abdallah Saffidine[5] and Torsten Schaub[3,5]

[1]*denkbares GmbH*
[2]*University of Würzburg*
[3]*University of Potsdam*
[4]*UP Transfer GmbH*
[5]*Potassco Solutions GmbH*

Abstract

Industrial product configuration requires robust, maintainable modeling environments that clearly separate domain knowledge from solving mechanisms. Declarative modeling addresses the maintainability aspects by focusing on valid configurations rather than control flow. Nonetheless, significant challenges remain when handling complex numerical calculations and generative component structures. To bridge the gap between expressive, general-purpose modeling languages and the need for specialized solving support, this paper introduces the *ShadowMaster*, a novel problem domain derived from real-world sun blind configuration. By isolating the difficulties of numerical reasoning and generative configuration into simplified models, we provide a systematic examination of these challenges. Furthermore, we present explicit declarative representations of the *ShadowMaster* domain using Coom, an open-source, solver-independent product configuration modeling language.

1. Introduction

Industrial product configuration [1] is a knowledge engineering problem before it is a solver problem [2, 3]. A configurator captures decisions that affect sales, engineering, production, and maintenance, and it evolves over time as products and business rules change. Because of this lifecycle, a company needs a modeling environment that domain experts and knowledge engineers can maintain, rather than just a collection of solver-specific tricks.

This naturally motivates declarative modeling. Configuration research has long emphasized representations that separate product domain knowledge from problem-solving knowledge [4, 5]. Rule-based configurators can be effective initially, but this separation becomes difficult when product knowledge is encoded together with control flow, rule ordering, and implementation-specific side effects. Declarative models instead aim to describe which configurations are valid, while leaving the search for such configurations to the solving system.

At the same time, industrial configuration problems often require forms of reasoning that are difficult to encode. Numerical calculations usually require additional machinery for SAT [6] and ASP [7] solvers when interacting with discrete choices [8, 9]. This becomes even more pronounced for non-linear formulas and real-valued engineering quantities. Component generation creates a different kind of difficulty because the solver has to reason about an unknown number of objects and references between them. Generative configuration has therefore been studied as a distinct issue in configuration research [10, 11] and recent work on ASP-based configuration investigates related questions for interactive and industrial-scale settings [12].

The resulting tension is that a modeling language should be general enough for knowledge engineers, while the underlying system still needs specialized solving support for the constructs that occur in practice. In this paper, we take a first step toward a systematic treatment of these challenges by introducing a novel problem domain. This domain describes the task of configuring sun blinds, which inspired us to dub it the *ShadowMaster*. The problem is simplified for this paper but derived from

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany



© 2026 This work is licensed under a "CC BY 4.0" license.

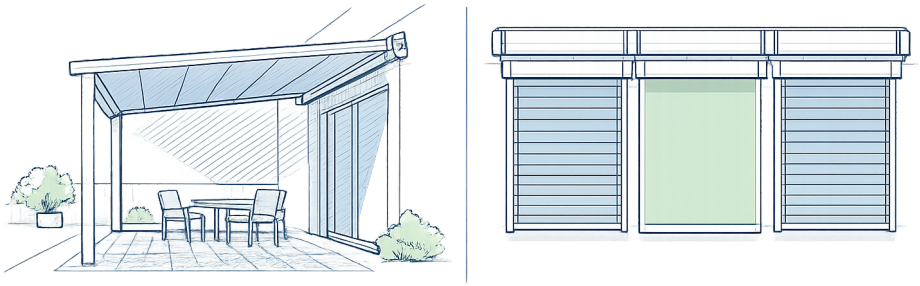


Figure 1: Representative sun blind systems motivating the *ShadowMaster* variants

real-world configuration work in industry such that the motivating modeling difficulties are preserved. Examples of the sun blinds encountered in practice are shown in Figure 1.

The rest of the paper is structured as follows. In Section 2, we start by describing two simplified versions of the *ShadowMaster*, each addressing only one of the two challenges. In Section 3, we provide explicit representations of both versions, specified in the product configuration language Coom [13]. We choose Coom because it offers a concrete, declarative notation for product structures, numerical features, and behavioral constraints and is not tied to a specific solver system. A workbench for experimenting with Coom product models, the CoomSuite [14], was introduced in [12]. Although initially targeted at ASP, it allows for the integration of different solver backends in a modular fashion. Both Coom and the CoomSuite are open source, so the examples can be inspected and experimented with.[1]

2. The *ShadowMaster* Problem

The problem is split into two deliberately simplified but complementary configuration tasks. The *ShadowMaster* 1500 isolates the geometric calculations for a single specific type of sun blind, a *pergola awning* (see left side of Figure 1), and challenges a configurator with non-linear numeric dependencies and feasibility constraints over derived values. The *ShadowMaster* 3000 abstracts from such geometry and instead focuses on generating and linking components for multiple neighboring blinds (see right side of Figure 1), requiring reasoning over an initially unknown number of objects and their associations. Together, the two variants capture capabilities that are central to industrial product configuration but difficult to implement robustly: non-linear calculations, component generation, cardinality reasoning, and structural consistency across references.

2.1. *ShadowMaster* 1500

We consider a simplified, two-dimensional side-view model of a pergola awning. The model is derived from a real installation problem but abstracts from several construction details; in particular, the awning is assumed to be straight, whereas some product variants use a curved guide rail.

Figure 2 shows a side-view of the geometry. Here, green symbols refer to constants and all other symbols refer to variables. The input parameters are

1. the *installation height* h_i,
2. the required *clearance height* h_c, and
3. the *input width* w_i.

The fixed product-specific offsets, obtained from the technical drawing, are

$$x_0 = 125, \qquad y_0 = 8.82, \qquad y_1 = 90, \qquad \rho = 48.4985\,, \tag{1}$$

[1]Note, however, that not all required features are currently implemented in the CoomSuite.

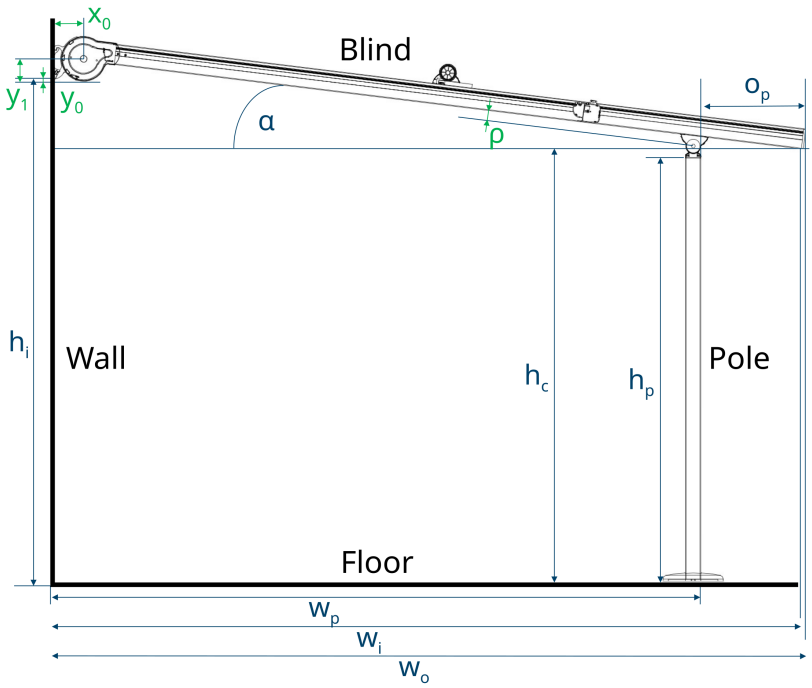


Figure 2: Simplified side view geometry of the *ShadowMaster* 1500

where x_0 is the wall-to-roller-axis offset, y_0 and y_1 are vertical cassette offsets, and ρ is the perpendicular offset between the post pivot and the awning reference axis. All quantities are assumed to be expressed in the same unit.

The central calculation of the awning is that of the *inclination angle* α. First, we define several auxiliary variables (omitted in Figure 2 for the sake of readability). The horizontal and vertical components Δx and Δy between the roller-axis reference point and the front clearance reference point are defined as

$$\Delta x = w_i - x_0, \qquad \Delta y = h_i - y_0 + y_1 - h_c\,. \tag{2}$$

Furthermore, we define

$$L = \sqrt{\Delta x^2 + \Delta y^2}\,. \tag{3}$$

The inclination angle α is then calculated as

$$\alpha = \operatorname{atan2}(\Delta y, \Delta x) + \arcsin\left(\frac{\rho}{L}\right)\,. \tag{4}$$

The first term is the inclination of the segment connecting the rear and front reference points. The second term corrects for the fact that the post pivot is not located on the awning reference axis but at a fixed perpendicular distance ρ. For the inclination calculation to be real-valued, the perpendicular offset must not exceed the distance between the reference points, that is, $0 \le \rho \le L$. In this simplified instance, the effective *order width* w_o is calculated as

$$w_o = w_i + 80\tan(\alpha)\,. \tag{5}$$

In addition, the configuration of the pergola awning may include a pole for extra stability. If a pole is used, its input value is the *pole width* w_p, that is, the distance of the pole from the wall. This must lie in the admissible range $0 \le w_p \le 4000$. Then, the *pole overhang* o_p is derived as

$$o_p = w_i - w_p\,, \tag{6}$$

and it must exceed the required minimum

$$o_p > 105\,. \tag{7}$$

Lastly, the *pole height* h_p must satisfy $1000 \leq h_p \leq 3200$ and is determined from the clearance height h_c and the inclination angle α, namely

$$h_p = h_c + o_p \tan(\alpha)\,. \tag{8}$$

2.2. *ShadowMaster* 3000

The *ShadowMaster* 3000 abstracts from the geometric calculation of a single blind and focuses on the configuration of several neighboring blinds. The difficulty is now the generation of a suitable object structure: given a number of blinds, the problem consists of determining the necessary number of engines together with consistent references between blinds and engines.

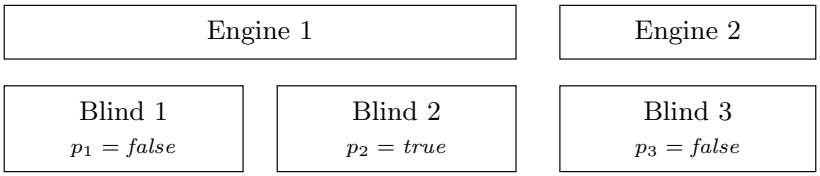


Figure 3: Schematic abstraction of neighboring blinds and their associated engines

Formally, as illustrated in Figure 3, the input to the problem is a fixed sequence of sun blinds

$$B = (b_1, \ldots, b_n) \tag{9}$$

with corresponding Boolean values

$$p_i \in \{true, false\} \quad \text{for } i \in \{1, \ldots, n\}\,, \tag{10}$$

where p_i denotes whether blind b_i is connected to the same engine as the previous blind. The first blind cannot refer to a predecessor, so

$$p_1 = false\,. \tag{11}$$

A solution consists of a generated sequence of engines $E = (e_1, \ldots, e_m)$ and an engine assignment

$$g : \{1, \ldots, n\} \to \{1, \ldots, m\}\,. \tag{12}$$

The intended meaning is that blind b_i is powered by engine $e_{g(i)}$. The assignment is constrained by the sequential order of the blinds such that the first blind is powered by the first engine, that is

$$g(1) = 1\,, \tag{13}$$

and, for $1 < i \leq n$ it holds that

$$p_i = true \Rightarrow g(i) = g(i-1) \tag{14}$$

$$p_i = false \Rightarrow g(i) = g(i-1) + 1\,. \tag{15}$$

Thus, setting p_i to false instantiates a new engine, while setting it to true attaches the current blind to the previous blind's engine.

The generated engines must also satisfy the engine cardinality constraints. In the simplified problem, each engine can serve between one and three blinds, that is

$$1 \leq |\{i \mid g(i) = j\}| \leq 3 \quad \text{for } j \in \{1, \ldots, m\}\,. \tag{16}$$

Consequently, once three consecutive blinds have been assigned to the same engine, the next blind cannot also be assigned to the same engine; otherwise the instance is infeasible.

Finally, the object references must be maintained in both directions. Each blind stores a single reference to its engine, and each engine stores the set of blinds assigned to it:

$$b_i.engine = e_{g(i)} \tag{17}$$

$$e_j.blinds = \{b_i \mid g(i) = j\}\,. \tag{18}$$

This bidirectional structure is important for knowledge engineers because it allows modular constraints to be attached to blinds, to engines, or to other functional groupings.

In practical product models, additional components may introduce further reference structures; for example, a shared housing could itself be a sequential component that can connect to a different subset of blinds. Furthermore, the problem can be posed as an optimization problem, for instance, by omitting the explicit p_i choices and asking for a valid configuration with a minimal number of engines.

3. Representing the *ShadowMaster* in Coom

We now illustrate how the two *ShadowMaster* variants can be represented in Coom. The encodings are intended to make the representational requirements of the problems explicit and to provide concrete models that can be inspected and experimented with in the CoomSuite and the Coom language environment. In particular, they show how the numerical dependencies of the *ShadowMaster* 1500 and the generated object structure of the *ShadowMaster* 3000 can be expressed in a declarative way. Note, however, that the CoomSuite solving backend does not yet support all the required features.

3.1. *ShadowMaster* 1500

The `product` keyword in Listing 1 initiates the Coom product model by defining the root-level features of the awning configuration. Each feature is defined by a type, a name, and an optional cardinality, which defaults to `1..1`. Numerical features may additionally have a range restricting the admissible values of the feature.[2] As in Section 2, the awning contains an optional `Pole`. The numerical features include the input variables `installationHeight`, `clearanceHeight`, and `inputWidth`, (corresponding to h_i, h_c, w_i), as well as the derived variables `inclination` and `orderWidth` (corresponding to α and w_o).

```
1  product {
2      0..1 Pole           pole
3           num            installationHeight
4           num            clearanceHeight
5           num            inputWidth
6           num            inclination
7           num            orderWidth
```

Listing 1: Root product declaration of the *ShadowMaster* 1500

Listing 2 is the continuation of Listing 1 and contains additional root-level features for the `material` of the awning cover with corresponding input variable `length` and derived variables for its `area` and `weight`. These additional features are not part of the formalization from Section 2 but illustrate further real-world product constraints that are typically derived from the core geometric calculations.

```
9           CoverMaterial  material
10          num            length
11          num            area
12          num            weight
13 }
```

Listing 2: Additional root features of the *ShadowMaster* 1500

[2] Note that Coom also allows specifying the unit and floating-point precision for numerical features but these are omitted here for brevity.

The `pole` and the `material` refer to separate types; both of which are shown in Listing 3. First, the `structure` keyword defines an abstract part `Pole`, which has numerical features for its `width`, `height`, and `overhang` (corresponding to w_p, h_p, and o_p from before). The admissible intervals for the `width` and `height` are given as ranges because they are intrinsic bounds of these features. Next, the `enumeration` keyword declares an enumeration `CoverMaterial` with a finite set of available options. Each option has a constant value with a corresponding `density`. This allows behavioral rules to compute the weight of the configured awning from the selected material later on.

```
15 structure Pole {
16     num    0-4000 width
17     num 1000-3200 height
18     num           overhang
19 }
20 enumeration CoverMaterial {
21     attribute num/gpqm density
22     _Default    = (  850 )
23     _Light      = (  599 )
24     _Acryl      = ( 1495 )
25     _AllWeather = ( 2450 )
26     _Kombi      = ( 2087 )
27 }
```

Listing 3: Pole structure and cover material enumeration of the *ShadowMaster* 1500

The `behavior` keyword describes constraints between variables of the model. Listing 4 shows the root-level behavioral knowledge of the awning. First, Lines 30-33 encode the constraint from (4) deriving the inclination angle α.[3] For the sake of readability, the fixed geometric constants from (1) are inlined directly. This is the central calculation of the *ShadowMaster* 1500 instance. Next, Line 34 encodes the constraint from (5). The remaining values for the `area` and total `weight` of the cover are subsequently derived from it. As mentioned above, these additional constraints are not part of the formalization from Section 2. Note how the selected cover variant enters the weight calculation through its material-specific `density` attribute.

```
30     require inclination =
31         atan((125 - inputWidth) / (installationHeight - 8.82 + 90 - clearanceHeight)) +
32         acos(48.4985 / sqrt((installationHeight - 8.82 + 90 - clearanceHeight) ^ 2
33                             + (125 - inputWidth) ^ 2))
34     require orderWidth = inputWidth + tan(inclination) * 80
35     require area = length * orderWidth / 1000000
36     require weight = area * material.density / 1000
37 }
```

Listing 4: Root-level behavior of the *ShadowMaster* 1500

Behavior blocks may also be scoped to a `structure` type. A block such as `behavior Pole` is evaluated separately for each `Pole` instance, and unqualified path expressions inside the block are interpreted relative to that instance. The keyword `root` refers back to the root product instance, which allows local constraints to access root-level features such as `inputWidth` or `clearanceHeight`. For instance, Listing 5 adds the `Pole` constraints (6)-(8).

```
40     require overhang = root.inputWidth - width
41     require overhang > 105
42     require height = root.clearanceHeight + tan(root.inclination) * overhang
43 }
```

Listing 5: `Pole`-specific behavior of the *ShadowMaster* 1500

[3] Equation (4) uses atan2 to state the geometry independently of quadrant conventions. Coom currently only provides atan, so Listing 4 uses a one-argument trigonometric formulation restricted to the intended geometric range.

3.2. *ShadowMaster* 3000

The *ShadowMaster* 3000 targets a different aspect of configurator capability. Expressing it declaratively is not trivial because the model must describe an initially unknown object structure. Component generation and the convenient declarative representation of such generated structures remain ongoing research problems and the language constructs for them are not yet final. The following representations should therefore be read as preliminary encodings, not as a final recommendation for the syntax.

Listing 6 shows the structural part of the model. It contains two types of parts: `Blind` and `Engine`. In the problem statement, the blind sequence is part of the instance. The cardinality `1..*` should therefore not be read as asking the solver to choose an arbitrary set of blinds. Rather, problem instances fix the number and order of `blinds`. The generated part of the model is the sequence of `engines` and the references between blinds and engines. The `reference` keyword is important here. A reference feature links to an already existing component elsewhere in the configuration instead of owning it. Thus, each blind has exactly one reference to an engine, while each engine connects to between one and three blinds. Note that the former stems from the nature of the function g from (12) and the latter from the cardinality defined in (16). The Boolean variable `withPrevious` corresponds to the variable p_i from (10). Additionally, the auxiliary numerical feature `engineBlindIndex` stores the position of a blind within its selected engine's local list of blinds.

```
1  product {
2    1..* Blind  blinds
3    1..* Engine engines
4  }
5  structure Blind {
6    bool                  withPrevious
7    num                   engineBlindIndex
8    reference 1..1 Engine engine
9  }
10 structure Engine {
11   reference 1..3 Blind  blinds
12 }
```

Listing 6: Product hierarchy of the *ShadowMaster* 3000

The behavioral part of the Coom model again uses local constraints. First, Listing 7 introduces two local definitions, `blindIndex` and `previousBlind`. These definitions only provide short names for expressions used in the subsequent constraints. The expression `index(this)` uses the special keyword `this`, which denotes the blind instance for which the `behavior Blind` block is currently evaluated. The function `index` returns the position of an instance within the feature of its parent object that owns the instance. Thus, `blindIndex` denotes the position of the current blind in the root-level `blinds` feature, and `previousBlind` abbreviates access to the preceding blind in that sequence. These constructs allow the model to relate a blind to its predecessor without iterating over all blinds. The subsequent lines encode the constraints specified in (11) and (13)-(15).

```
14 behavior Blind {
15   define blindIndex = index(this)
16   define previousBlind = root.blinds[blindIndex - 1]

18   condition blindIndex = 0
19   require withPrevious = false && index(engine) = 0

21   condition withPrevious = true
22   require engine = previousBlind.engine

24   condition blindIndex > 0 && withPrevious = false
25   require index(engine) = index(previousBlind.engine) + 1
```

Listing 7: Constraints for sequential engine assignment

Next, Listing 8 continues the `behavior` block from Listing 7 and constrains the position of the current blind within its engine's list of blinds. A blind starting a new sequence is always assigned `engineBlindIndex` 0, and otherwise it increments the `engineBlindIndex` of the previous blind by 1. Note that in this way the `engineBlindIndex` is always a value between 0 and 2 as follows from the other constraints and the cardinality `1..3` in Listing 6.

```
27    condition withPrevious = false
28    require engineBlindIndex = 0

30    condition withPrevious = true
31    require engineBlindIndex = previousBlind.engineBlindIndex + 1
32  }
```

Listing 8: Constraints for blind index in engine reference

Lastly, the two constraints in Listing 9 encode the requirement from (17) and (18) that the references are bidirectional.

```
34  behavior Blind  { require engine.blinds[engineBlindIndex] = this }
35  behavior Engine { require blinds.engine = this }
```

Listing 9: Constraints for bidirectional references

4. Discussion

We introduced a novel problem domain, the *ShadowMaster*, which describes configurations of different types of sun blinds. The *ShadowMaster* 1500 characterizes the geometric aspects of a pergola awning and involves challenging non-linear calculations. The *ShadowMaster* 3000 defines a system of sun blinds with corresponding engine assignments and deals with component generation, cardinality reasoning, and structural consistency across references. We presented a mathematical description of the two problems and corresponding preliminary Coom product models.

Both problems are highly non-trivial even when considered by themselves. In industrial settings, however, combinations of the above challenges are common. It remains an open research question to find compact and maintable represenations as well as efficient and flexible solving approaches. For instance, in ASP even the treatment of floating-point numbers, let alone of non-linear functions such as trigonometric ones, is already non-trivial. Other approaches, such as constraint programming [15], naturally deal with these kinds of numerical domains but do not natively support object generation and references. Although the simplified *ShadowMaster* 3000 has a natural upper bound on the number of engines by the given sequence of blinds, configurators with open-ended component generation may require non-standard incremental approaches.

While this paper focuses on introducing and describing the problem domain, implementing the necessary solving technologies is in active development. Currently, this is done in the context of the CoomSuite using ASP; however, a study and comparison with other approaches would be insightful. For this purpose, we plan to create scalable and modular benchmark versions of the problems introduced here. One straightforward but more complex extension would be a combination of the two isolated problems from Section 2. For example, instead of a single pergola awning, a sequence of awnings could be configured, each with possibly different requirements regarding size, material, and other factors.

Acknowledgments

This work was partly funded by ZIM BMBK grants KK5291307GR4 and KK5394902GR4.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini and ChatGPT in order to: Abstract drafting, Paraphrase and Reword, Generate images (figure 1). After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] R. Comploi-Taupe, A. Falkner, Product Configuration - An Introduction for Practitioners, Springer Briefs in Computer Science, Springer-Verlag, 2026. doi:10.1007/978-3-031-61874-1.

[2] 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.

[3] G. Friedrich, D. Jannach, M. Stumptner, M. Zanker, Knowledge engineering for configuration systems, in: [16], 2014, pp. 139–155. doi:10.1016/B978-0-12-415817-7.00011-6.

[4] L. Hotz, A. Felfernig, M. Stumptner, A. Ryabokon, C. Bagley, K. Wolter, Configuration knowledge representation and reasoning, in: [16], 2014, pp. 41–72. doi:10.1016/b978-0-12-415817-7.00006-2.

[5] T. Soininen, I. Niemelä, Developing a declarative rule language for applications in product configuration, in: G. Gupta (Ed.), Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99), volume 1551 of *Lecture Notes in Computer Science*, Springer-Verlag, 1999, pp. 305–319. doi:10.1007/3-540-49201-1_21.

[6] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability - Second Edition, volume 336 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2021.

[7] V. Lifschitz, Answer Set Programming, Springer-Verlag, 2019. doi:10.1007/978-3-030-24658-7.

[8] M. Ostrowski, T. Schaub, ASP modulo CSP: The clingcon system, Theory and Practice of Logic Programming 12 (2012) 485–503. doi:10.1017/S1471068412000142.

[9] Y. Lierler, Constraint answer set programming: Integrational and translational (or SMT-based) approaches, Theory and Practice of Logic Programming 23 (2023) 195–225. doi:10.1017/S1471068421000478.

[10] M. Stumptner, G. Friedrich, A. Haselböck, Generative constraint-based configuration of large technical systems, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12 (1998) 307–320. doi:10.1017/s0890060498124046.

[11] G. Fleishanderl, G. Friedrich, A. Haselbock, H. Schreiner, M. Stumptner, Configuring large systems using generative constraint satisfaction, IEEE Intelligent Systems and their Applications 13 (1998) 59–68.

[12] J. Baumeister, K. Herud, M. Ostrowski, J. Reutelshoefer, N. Rühling, T. Schaub, P. Wanko, Towards industrial-scale product configuration, Theory and Practice of Logic Programming (2026) 1–31. doi:10.1017/S1471068426100404.

[13] J. Baumeister, K. Herud, J. Reutelshöfer, V. Belli, Coom language, https://www.coom-lang.org/, 2025.

[14] J. Baumeister, K. Herud, M. Ostrowski, J. Reutelshöfer, N. Rühling, T. Schaub, P. Wanko, CoomSuite, https://github.com/potassco/coom-suite, 2025.

[15] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier Science, 2006.

[16] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen (Eds.), Knowledge-Based Configuration: From Research to Business Cases, Elsevier/Morgan Kaufmann, 2014. doi:10.1016/C2011-0-69705-4.

Broken Retail States as Configurable Objects in Retail Shelf Systems

Kateryna Czerniachowska

Wroclaw University of Economics and Business, Komandorska St 118/120, 53-345, Wroclaw, Poland

Abstract

Modern retail environments rely on complex interconnected information systems supporting inventory management, shelf space allocation, replenishment, planogram generation, and promotional operations. However, operational retail data frequently contains inconsistencies. Existing retail optimization approaches commonly assume that input data is structurally valid, which limits their applicability in real-world environments characterized by dynamically evolving retail states. This paper proposes a configuration-oriented framework for representing and repairing inconsistent retail operational data. The proposed approach interprets retail inconsistencies as invalid configuration states within a constrained retail configuration space. To formalize this perspective, the paper introduces the concept of a Broken Retail State. The framework adopts a layered architecture consisting of a Raw Retail Data Layer, a Consistency Configuration Layer, a Valid Retail Representation Layer, and a downstream Optimization and Decision Layer. The central contribution of the framework is the Consistency Configuration Layer, which performs inconsistency diagnosis and controlled retail state transformation through the application of configuration operators. The proposed operators include completion, removal, substitution, synchronization, and consolidation mechanisms responsible for transforming invalid retail states into structurally valid ones. Additionally, the paper introduces a taxonomy of Broken Retail States and formalizes retail consistency restoration as a configuration transformation process operating over constrained retail state spaces. Several illustrative scenarios are presented to demonstrate the applicability of the framework to common retail inconsistencies.

Keywords

Shelf space allocation, retail configuration, data inconsistency, retail data quality, broken retail states, configuration framework, operational retail data, intelligent retail systems.

1. Introduction

Retail environments increasingly rely on data-driven decision-making processes to support operational activities such as inventory management, planogram generation, shelf space allocation, replenishment, and demand forecasting [1–3]. Modern retail systems integrate information originating from multiple heterogeneous sources, including point-of-sale systems, enterprise resource planning platforms, warehouse management systems, shelf monitoring technologies, and promotional management applications. Although these systems enable large-scale optimization and automation, they also introduce specific challenges related to data consistency and structural integrity. In practice, retail data frequently contains incomplete assignments, contradictory records, duplicated entities, outdated planograms, phantom inventory states, and inconsistencies between physical shelf observations and digital representations.

Such inconsistencies do not originate exclusively from technical integration problems. Retail environments involve multiple stakeholders, including assortment planners, category managers, store employees responsible for shelf replenishment, warehouse workers, logistics personnel, and store managers. These actors interact with different information systems and maintain

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany
kateryna.czerniachowska@ue.wroc.pl (K. Czerniachowska)
0000-0002-1808-6020 (K. Czerniachowska)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

different perspectives on retail operations. Consequently, inconsistencies may emerge due to manual data entry errors, differences in operational priorities, incomplete execution of planned shelf layouts, usability limitations of retail software interfaces, or discrepancies between strategic retail plans and physical store execution. The resulting inconsistencies are therefore often socio-technical in nature, involving both information systems and human operational processes.

Existing research on Shelf Space Allocation Problems (SSAP), assortment optimization, and retail analytics typically assumes that input data is valid, consistent, and operationally reliable [2–6]. Under this assumption, optimization methods focus primarily on maximizing sales or profitability. However, real-world retail environments rarely satisfy these assumptions. Optimization processes executed on inconsistent retail data may produce infeasible or non-interpretable solutions. Consequently, inconsistencies in retail operational data represent a data quality concern. It is a structural problem that can reduce the feasibility and reliability of optimization results.

From the perspective of configuration research, retail data inconsistencies can be interpreted as invalid configuration states [7–9]. Retail systems consist of interconnected entities such as products, shelves, facings, inventory records, replenishment policies, and promotional assignments that must satisfy a variety of physical, semantic, temporal, and business constraints. For instance, a product may be assigned to a shelf section that lacks sufficient physical space, or a replenishment system may generate deliveries for products that have already been discontinued in the assortment database. In case of violation of constraints, retail systems may enter inconsistent operational states that decision-support systems cannot process reliably. In this context, restoring retail data consistency can be treated as a configuration problem in which inconsistent states are transformed into valid operational representations using configuration operators and consistency rules.

This paper proposes a layered configuration framework for repairing inconsistent retail data states, building upon concepts from constraint-based configuration and inconsistency diagnosis while addressing challenges that have received limited attention in retail optimization research. The proposed framework introduces the concept of a Broken Retail State, defined as a retail operational representation violating one or more constraints. Unlike conventional approaches focused directly on optimization, the proposed framework establishes a dedicated consistency-configuration layer responsible for detecting, classifying, and transforming inconsistent retail states into valid and traceable retail representations. The framework is a preprocessing mechanism that enables downstream optimization processes. As a result, analytical systems can operate on feasible and reliable data.

This work makes conceptual and theoretical contributions. The paper formalizes retail inconsistencies as configurable entities within a retail configuration space. It then introduces a taxonomy of broken retail states observed in large-scale retail environments and defines a set of configuration operators capable of repairing invalid retail states and transforming them into consistent operational representations. Finally, the paper presents a layered architecture that separates raw retail data, consistency configuration processes, and downstream optimization mechanisms.

The remainder of the paper is organized as follows. Section 2 discusses related work in retail optimization, configuration systems, and data consistency management. Section 3 introduces the formal definition of Broken Retail States and the proposed taxonomy of retail inconsistencies. Section 4 presents the layered configuration framework. Section 5 defines the consistency-configuration layer, configuration operators, and state transformation mechanisms. Section 6 provides illustrative retail inconsistency scenarios and configuration examples. Section 7 presents the discussion. Finally, Section 8 concludes the paper and outlines directions for future research.

2. Related Literature

Research related to retail shelf management, configuration systems, and data inconsistency handling spans multiple domains. Typically, the relevant domains include retail optimization, constraint-based configuration, knowledge representation, and data consistency management. However, substantial progress has been achieved in Shelf Space Allocation Problems (SSAP) and retail decision-support systems.

2.1. Retail Shelf Space Allocation and Retail Optimization

Retail shelf space allocation has been extensively studied in operations research and retail analytics literature. Early research focused primarily on the relationship between shelf exposure and product sales, while later studies introduced mathematical optimization methods for allocating limited retail shelf resources [1,6,10,11].

Yang and Chen [6] introduced one of the earlier comprehensive optimization approaches to shelf management, emphasizing the importance of operational constraints in practical retail environments [6]. Subsequent research extended shelf allocation models toward more sophisticated formulations involving assortment planning, replenishment integration, category management, and planogram generation.

Bianchi-Aguiar et al. [1] provided a comprehensive review and classification framework for retail shelf space planning problems, demonstrating the increasing complexity of shelf allocation models and highlighting the importance of operational constraints in modern retail systems [1]. Their work systematized the literature surrounding shelf space planning and identified the lack of unified representations and benchmark formulations.

Hübner et al. [2] further extended shelf space allocation research by integrating shelf dimensioning and product allocation within a unified optimization framework [2]. Their work demonstrated that shelf configuration and product allocation are strongly interdependent processes. Similarly, Flamand et al. [5] investigated store-wide shelf allocation strategies supporting impulse buying and category interactions [5].

Additional contributions were provided through the development of heuristic frameworks integrating practical retail shelf constraints, such as facings, capping, and nesting methods, into shelf allocation decision support. These studies highlighted the importance of realistic operational conditions and demonstrated that retail shelf allocation problems frequently involve highly constrained decision spaces [12,13]. Further research addressed shelf-space allocation while considering vertical positioning effects, product visibility, and merchandising constraints. The findings indicated that shelf allocation decisions depend on optimization objectives and on structural and contextual relationships between retail entities [14].

The next research emphasized the complexity of planogram symmetry, shelf segmentation, and product clustering in retail configuration environments. The authors introduced hyper-heuristic approaches for handling large constrained shelf allocation problems involving multiple interconnected operational rules [15].

Recent studies additionally incorporate replenishment and assortment planning into integrated retail optimization systems. For example, integrated decision-support frameworks combining assortment optimization, shelf allocation, and replenishment planning were proposed to improve operational coordination across retail processes [3].

Integrated retail planning approaches have also gained increasing attention in recent years. Hübner and Schaal [4] investigated integrated shelf planning and replenishment coordination mechanisms supporting large-scale retail operations. Their work demonstrated that shelf allocation decisions strongly influence inventory management and operational logistics [4]. Borin et al. [16] proposed integrated category-management models considering profitability and

shelf-space utilization simultaneously. Their work became an important foundation for later integrated retail optimization studies [16].

Another important direction concerns computer-vision-assisted retail monitoring. Chadha [17] explored automated shelf recognition and planogram verification systems using deep learning. Their results demonstrated that discrepancies between detected shelf states and stored retail representations are common in operational retail environments, reinforcing the need for consistency restoration mechanisms [17].

Mesquita et al. [18] proposed real-time synchronization architectures integrating computer vision, RFID technologies, and inventory systems. Their work emphasized that synchronization delays and heterogeneous data streams frequently produce inconsistent retail representations requiring automated consistency restoration mechanisms [18].

Additional studies have investigated sustainability-aware shelf optimization and environmentally adaptive retail systems. Yu et al [19] assessed the implications of a fixed carbon tax, tradable emission allowances, and emission pricing and incorporated sustainability metrics into shelf allocation optimization. Their work demonstrated that sustainability-related business constraints introduce additional configuration dependencies capable of generating operational inconsistencies [19].

Recent studies increasingly combine shelf allocation with broader retail analytics and operational coordination systems. Integrated approaches simultaneously addressing assortment planning, replenishment synchronization, category management, and shelf optimization have become particularly important in large-scale retail environments characterized by dynamically changing assortments and operational uncertainty.

Despite the maturity of SSAP research, most existing models assume that retail operational data is consistent, synchronized, and structurally valid. In practice, however, retail systems frequently contain conflicting assignments, outdated planograms, duplicate records, phantom inventory states, and synchronization delays. Such inconsistencies are rarely formalized explicitly within retail optimization literature.

2.2. Configuration Systems and Constraint-Based Reasoning

Configuration research traditionally focuses on constructing valid product or system instances from predefined components and constraints. In such systems, consistency maintenance plays a fundamental role because invalid configurations violate structural or semantic rules that are defined within the configuration space.

Felfernig et al. [7] extensively investigated knowledge-based configuration systems, recommender-supported configuration engineering, and inconsistency diagnosis in constraint-based environments. The authors emphasized the importance of explanation mechanisms, configuration repair, and intelligent recommendation processes in complex configuration spaces[7]. Similarly, Felfernig et al. [8] introduced diagnosis mechanisms for identifying conflicting constraints within over-constrained systems [8].

Pereira et al. [9] analyzed methods for representing, learning, and exploring configuration spaces in software systems. Their work demonstrated that complex configurable systems frequently contain large numbers of interdependent constraints and configuration variants. Although their research focuses on software systems, the concept of configuration spaces and invalid configurations provides important theoretical foundations for modeling retail operational states as constrained configuration environments [9].

Research on feature models and configurable systems additionally highlights the importance of constraint propagation, dependency management, and explainability in complex configuration environments. The concepts of it are highly relevant in retail systems where products, shelves, planograms, promotions, and replenishment structures form strongly interconnected operational networks.

Further research investigated practical shelf-space allocation under realistic merchandising conditions involving horizontal and vertical grouping, capping, nesting, and multi-shelf constraints. The work showed that retail shelf allocation problems naturally contain large numbers of interacting constraints and operational dependencies, making them structurally similar to complex configuration systems [13].

Investigation of merchandising rules is another interesting direction in retailing. Researchers analyzed the relationship between product categorization, shelf positioning, brand visibility, and customer perception. Their work demonstrated that retail shelf configurations are governed by physical constraints and contextual business rules [20].

Constraint satisfaction and diagnosis methods can be interpreted as over-constrained or invalid configuration states. In this context, diagnosis mechanisms identify violated constraints, while repair operators transform inconsistent states into admissible configurations.

However, existing configuration research primarily focuses on configurable products, software systems, or feature models. The underlying principles of configuration theory can also be applied to operational retail states composed of interconnected entities and business constraints.

2.3. Data Consistency and Knowledge Integration

Research on data consistency management and knowledge integration provides important foundations. Distributed information systems frequently experience inconsistencies caused by asynchronous updates, heterogeneous schemas, incomplete synchronization, or conflicting representations across integrated systems.

In retail environments, these problems are amplified by the coexistence of multiple operational platforms such as inventory systems, point-of-sale systems, warehouse management systems, and shelf monitoring technologies. Consequently, inconsistencies emerge at the database level as well as at the operational level.

The relevance of structured operational representations is also visible in advanced shelf-space allocation literature. Realistic shelf-space allocation requires the integration of multiple spatial, merchandising, and category-management constraints [21]. The work emphasized the importance of representing shelf structures as interconnected operational systems rather than isolated allocation variables [21].

Several researchers analyzed allocation consistency in both retail stores and distribution centers while considering multi-oriented capping constraints and operational storage dependencies. Their work further illustrates that retail operational structures inherently contain complex interdependencies capable of generating structurally inconsistent states [22].

Additional research demonstrated that retailers frequently impose predefined allocation restrictions related to product type, positioning strategy, package characteristics, or brand visibility. Such restrictions significantly increase the likelihood of operational inconsistencies when multiple systems simultaneously modify retail representations [23].

Recent configuration research has increasingly focused on constraint satisfaction, configuration repair, and reconfiguration processes in environments where system states evolve over time. Dynamic configuration environments require transforming invalid or outdated configurations into admissible states while preserving as much of the original configuration as possible.

Research on knowledge-based configuration systems provides a formal perspective on consistency management that is directly relevant to the proposed Broken Retail States concept. Felfernig et al. [24] defined a configuration as valid only when it satisfies all constraints encoded in the underlying knowledge base and distinguished between consistent and inconsistent system states through declarative constraint reasoning. Their consistency-based diagnosis framework further demonstrates that violations of configuration constraints can be automatically identified,

localized, and explained by analyzing conflicts between expected and observed system states [24]. This view closely aligns with the proposed Consistency Configuration Layer, where Broken Retail States represent retail system states that violate semantic, operational, or business constraints governing correct product, inventory, pricing, or process configurations.

Lambrix [25] argued that knowledge-based systems require mechanisms for detecting inconsistencies and for repairing incomplete or incorrect knowledge structures. Ontology debugging distinguishes between semantic defects, modeling defects, and missing knowledge, emphasizing that incorrect or incomplete domain representations can lead to erroneous reasoning and decision-making. The proposed framework for ontology repair further formalized defect detection and correction as a reasoning problem, reinforcing the view that maintaining consistent knowledge requires continuous validation against domain constraints [25].

Recent AI-oriented studies investigated adaptive retail shelf systems capable of responding dynamically to changes in demand, inventory, and customer behavior. Jayakrishnan [26] analyzed machine-learning and AI-based approaches supporting automated merchandising, demand-aware shelf allocation, and customer-centric planogram generation. Their review demonstrated that intelligent shelf systems increasingly depend on large-scale operational data integration and real-time adaptation mechanisms [26].

Interesting research direction concerns reinforcement-learning-based inventory optimization for perishable goods. Nomura et al. [27] investigated dynamic pricing and ordering policies using deep reinforcement learning and proximal policy optimization. Their findings demonstrated that reinforcement-learning-based approaches can significantly reduce computation time while maintaining near-optimal performance in complex perishable inventory environments, highlighting the potential of AI-driven decision systems for reducing waste and improving retail profitability [27].

Similarly, Wu et al. [28] introduced reinforcement-learning approaches for adaptive shelf allocation under continuously evolving inventory and demand conditions. The authors showed that dynamic retail environments generate frequent inconsistencies between digital plans and physical shelf states, especially when autonomous allocation decisions are performed continuously [28].

Interactive ontology debugging further extends consistency management by focusing on efficient fault localization in knowledge bases. Shchekotykhin et al. [29] proposed an interactive diagnosis framework that treats ontology inconsistencies as violations of logical axioms and progressively identifies the underlying faulty constraints through user-guided reasoning. Their entropy-based query strategy minimized the effort required to isolate the source of inconsistencies while maintaining formal correctness. The work highlighted that consistency management includes identifying the specific semantic rules responsible for those states [29]. Similarly, the proposed Consistency Configuration Layer aims to detect Broken Retail States and to support the localization of semantic, temporal, and operational constraint violations that generate inconsistencies in retail operations.

The proposed taxonomy of Broken Retail States is also consistent with research on database consistency and repair. Bertossi et al. [30] and colleagues showed that inconsistent databases can contain multiple valid repair alternatives, with repair strategies determined by integrity constraints and preference relations among conflicting facts . Preferred repair frameworks formalize inconsistency as a violation of database constraints and define principled mechanisms for restoring valid database states [30]. This perspective is particularly relevant for retail environments, where conflicting inventory records, product attributes, or pricing information may admit several possible corrections that must be resolved according to business priorities and operational policies.

Declarative knowledge representation and reasoning approaches, particularly Answer Set Programming (ASP), provide a well-established computational foundation for consistency

management in complex systems. As reviewed by Falkner et al. [31], ASP has been successfully applied in industrial domains including configuration, diagnosis, planning, and repair, where system states are represented as sets of constraints and valid solutions are generated through automated reasoning. A key advantage of ASP is its ability to identify inconsistent configurations, explain conflicting requirements, and compute valid repairs while optimizing according to business constraints [31]. These capabilities closely align with the objectives of the proposed Consistency Configuration Layer, suggesting that Broken Retail States can be interpreted as constraint violations amenable to declarative reasoning and automated diagnosis.

Kaminski and Schaub [32] developed a rigorous theoretical foundation for grounding in ASP. Grounding is the process of converting logic programs containing variables into equivalent propositional (variable-free) programs that ASP solvers can process. They provided a semantic and algorithmic framework that closely matches how practical grounders such as gringo operate [32].

Gonçalves et al. [33] provided a comprehensive overview of forgetting (also called variable elimination) in ASP. Forgetting is the operation of removing atoms, variables, or symbols from a logic program while preserving as much of its original meaning as possible over the remaining vocabulary. The authors systematically compared the many forgetting operators proposed in the literature, analyzed their properties, computational complexity, and applicability, and offered guidance on selecting the most appropriate operator for different applications [33].

Recent literature demonstrates several important trends, including increasing integration of retail planning processes, growing adoption of AI-driven shelf optimization, expansion of dynamic and adaptive merchandising systems, and increasing importance of explainability and sustainability. The studies indicate that retail operational environments naturally generate conflicting assignments, synchronization delays, inconsistent planograms, incompatible product placements, and structurally infeasible shelf representations.

Traditional data-cleaning approaches generally focus on removing corrupted records, filling missing values, or resolving duplicates. However, these approaches often treat inconsistencies as defects rather than structured operational states embedded within broader business processes. Consequently, there remains a significant gap between optimization-oriented retail research and the operational reality of inconsistent retail environments.

3. Broken Retail State Formalization

Retail systems rely on interconnected data describing products, shelves, inventory levels, replenishment operations, pricing structures, promotions, and customer-facing planograms [34–37]. Such retail entities are continuously updated by different information systems and physical operational processes. Retailers' transformation into digital marketplaces requires retail information systems to support their dual role as both marketplace operators and direct sellers. Implementation of retail information systems enables multi-sided digital marketplace business models [38].

Joint optimization of product selection and shelf space allocation in omnichannel retailing is very important. It incorporates the showroom effect, where customers' in-store experiences influence their online purchasing decisions. Chen et al. [39] proposed an optimization model that simultaneously determines online and offline assortments and shelf space allocation to maximize overall profitability. The authors demonstrated that integrated decision-making outperforms separate optimization approaches and provides valuable managerial insights for improving omnichannel retail performance [39].

While existing configuration research primarily addresses engineering and product configuration domains [24], the present study extends these principles to omnichannel retail

operations by conceptualizing retail inconsistencies as knowledge-based configuration violations that can be systematically detected and classified.

Because retail environments are highly distributed and constantly changing, inconsistencies often emerge between the physical state of a store and its digital representation. For example, a retail system may indicate that a product is available in backroom inventory while shelf-monitoring sensors simultaneously report an empty shelf. Similarly, promotional systems may assign discounted products to planograms that no longer reflect the actual shelf layout used in stores. Most of the inconsistencies may originate from synchronization delays, incomplete updates, conflicting operational rules, manual interventions, or errors introduced during data integration processes.

Conventional retail optimization approaches usually treat retail inconsistencies as isolated data quality issues handled through preprocessing or filtering. In contrast, this work interprets inconsistent retail representations as invalid configuration states. From a configuration perspective, a retail environment may be understood as a structured space consisting of entities, relationships, and operational constraints.

A retail state is considered valid only if all relevant constraints are satisfied simultaneously. When these constraints are violated, the resulting retail representations become structurally inconsistent; in this work, such states are referred to as Broken Retail States. In practice, many of these inconsistencies coexist and may propagate across multiple retail systems simultaneously.

A Broken Retail State (BRS) is defined as a retail operational representation violating at least one structural, semantic, temporal, or operational constraint within the retail configuration space. Inconsistencies are interpreted as invalid configurations within the retail space rather than merely corrupted data records.

A retail state $S$is considered valid if:

$$\forall c_i \in C: c_i(S) = true$$

A BRS occurs whenever at least one constraint is violated:

$$\exists c_i \in C: c_i(S) = false$$

The framework distinguishes several categories of constraints that maintain operational consistency in retail systems.

Structural constraints define physical and logical placement conditions related to shelves, products, and spatial assignments. They ensure that product allocations satisfy shelf dimensions, category structures, and physical compatibility rules. Violations of structural constraints may lead to physically infeasible shelf configurations. An example structural constraint is shelf capacity limitation:

$$\sum_{i=1}^{n} w_i x_i \leq W_s$$

where:

- w_i denotes the width of product i,
- x_i denotes the assignment decision variable,
- W_s denotes the available shelf width.

Semantic constraints preserve logical consistency between retail entities and their attributes. Their purpose is to maintain compatibility between product properties, inventory records, and promotional assignments across integrated systems. Common examples include consistency between product categories and shelf zones, consistency between inventory records and observed shelf presence, and compatibility between promotional states and planogram assignments.

Temporal constraints govern synchronization between dynamically changing retail processes and digital representations. Retail systems continuously evolve due to replenishment

operations, assortment changes, seasonal promotions, and inventory movements. Consequently, timing discrepancies frequently introduce inconsistencies. Therefore, some examples of such constraints include outdated planograms, delayed inventory synchronization, expired product assignments, and stale replenishment schedules.

Operational constraints describe business and execution rules required for daily retail functioning. They ensure compatibility between replenishment operations, pricing systems, promotional strategies, and physical shelf accessibility. Among the examples are minimum facing requirements, replenishment feasibility, price synchronization, and promotional placement obligations.

The proposed framework introduces a taxonomy of Broken Retail States commonly observed in large-scale retail environments. It groups inconsistencies according to the type of constraint that has been violated.

Structurally broken states are associated with violations of physical or logical placement rules. In practice, these problems often appear during assortment updates, shelf resets, or automated planogram generation. Typical examples include shelf-capacity overflows, duplicate product assignments, missing shelf allocations, and products placed in incorrect hierarchy segments. Such inconsistencies may produce shelf configurations that are physically infeasible or operationally difficult to maintain.

Semantic broken states arise when related retail entities contain contradictory or incompatible information. They are especially common in distributed retail information systems where data originates from multiple independent sources. For instance, inventory systems may report stock availability while shelf-monitoring systems detect empty shelves, creating phantom inventory situations. Similar issues may occur when different systems assign conflicting product attributes or incompatible category labels.

Temporal broken states emerge from delays or synchronization mismatches between operational processes and their digital representations. Because retail environments continuously evolve through replenishment activities, assortment changes, and promotional updates, timing inconsistencies are difficult to avoid entirely. Common examples include outdated planograms, expired product placements, delayed replenishment synchronization, and promotional configurations that no longer correspond to current store conditions.

Operational broken states affect the execution of everyday retail processes and therefore have a direct impact on store performance and customer experience. These inconsistencies may involve replenishment conflicts, pricing mismatches between systems and shelf labels, insufficient product facings, or inaccessible replenishment paths that prevent efficient shelf restocking. In many cases, operational inconsistencies emerge as a consequence of unresolved structural, semantic, or temporal problems.

The proposed taxonomy is also consistent with broader classifications of integrity constraints discussed in database systems, knowledge integration, and data quality management literature. Structural constraints correspond to schema-level and physical consistency requirements. Semantic constraints are closely related to business-rule consistency and ontology alignment. Temporal constraints reflect synchronization and temporal database consistency requirements. Operational constraints capture process-oriented business restrictions governing executable retail operations. Consequently, the proposed taxonomy adapts established consistency concepts to the retail configuration domain.

The designed framework treats BRS as configurable objects subject to transformation and repair operations. The framework preserves inconsistent states as explicitly identifiable operational configurations. This approach supports explainable diagnosis, traceable repair procedures, and more systematic consistency restoration.

4. Retail Configuration Framework

The proposed Retail Configuration Framework establishes a structured approach for representing, diagnosing, and transforming inconsistent retail operational states. The framework is designed to support consistency restoration in large-scale retail environments. Often, such retail spaces are characterized by heterogeneous data sources, asynchronous operational processes, and continuously evolving shelf configurations. Unlike traditional retail optimization systems that assume the existence of valid input data, the proposed framework explicitly incorporates inconsistency handling as a dedicated configuration problem.

The framework adopts a layered architecture separating raw operational representations, consistency restoration processes, and downstream analytical mechanisms. In such a way, it isolates the inconsistent data management from optimization procedures. Each step preserves traceability and explainability of configuration transformations. The framework consists of four conceptual layers, illustrated in Figure 1.



Figure 1: Framework Layers

The primary objective of the framework is to correct retail representations suitable for downstream optimization and analytical processing. The framework, therefore, operates as a preprocessing mechanism transforming invalid retail states into feasible and reliable data.

The proposed framework pursues four principal objectives, including representation of inconsistent retail operational states, identification and classification of constraint violations, transformation of invalid states into consistent retail representations, and preservation of traceable and explainable configuration processes. Consequently, inconsistent states are preserved as explicitly represented configurations subject to diagnosis.

The Raw Retail Data Layer contains heterogeneous operational representations collected from distributed retail systems. Data sources may include point-of-sale systems, inventory management systems, warehouse management platforms, shelf monitoring technologies, replenishment logs, planogram databases, and promotional management systems. This layer intentionally preserves inconsistencies, missing values, contradictory assignments, and synchronization conflicts originating from operational retail processes. It therefore represents the operational retail environment in its naturally evolving form.

The Consistency Configuration Layer constitutes the central component of the framework and represents its primary scientific contribution. This layer interprets inconsistent retail states as invalid configurations within the retail configuration space. Under its responsibilities are the detection of constraint violations, classification of BRS, application of configuration operators, generation of consistent retail representations, and preservation of transformation traceability. The layer operates using configuration rules, consistency constraints, and transformation operators defined over the retail configuration space. Unlike conventional preprocessing approaches that eliminate inconsistent records, the proposed layer explicitly models inconsistencies as configurable entities subject to controlled transformation processes.

The Valid Retail Representation Layer stores retail states satisfying all required consistency constraints. States within this layer constitute admissible retail configurations suitable for downstream optimization and analytical processing. The resulting representations provide structural consistency, semantic compatibility, temporal synchronization, and operational

feasibility. This layer, therefore, defines the feasible retail configuration space generated by the consistency restoration process.

The Optimization and Decision Layer contains downstream optimization and decision-support mechanisms and operates on valid data configurations. These mechanisms may include shelf space allocation optimization, assortment planning, replenishment optimization, sustainability-aware retail analytics, pricing optimization, and demand forecasting.

The proposed framework provides valid input data enabling reliable operation of analytical systems. The framework models the retail environment as a constrained configuration space composed of entities, relationships, and operational rules. Formally, the retail configuration space (RCS) is defined as:

$$RCS = (E, R, C, O)$$

where:

- E denotes the set of retail entities,
- R denotes the set of relationships between entities,
- C denotes the set of consistency constraints governing admissible retail states.
- O denotes the set of configuration operators.

Retail entities may include products, shelves, categories, inventory records, planograms, promotional assignments, replenishment schedules, or pricing structures. Relationships define interactions between entities, such as product placement, category hierarchy, or inventory allocation. Constraints define admissible operational conditions within the retail environment. Configuration operators define transformation mechanisms capable of converting invalid retail states into structurally consistent representations. Valid retail states correspond to admissible configurations satisfying all constraints within the configuration space.

The framework adopts a state transformation perspective in which retail consistency restoration is modeled as a sequence of configuration transitions between invalid and valid operational states. Given a BRS, the framework applies a transformation process:

$$T: BRS \rightarrow CRS$$

where:

- BRS denotes a broken retail state,
- CRS denotes a consistent retail state,
- T denotes a configuration transformation process.

Typically, transformation processes may involve completion of missing assignments, removal of contradictory relationships, synchronization of temporal records, substitution of invalid placements, and consolidation of conflicting operational representations. The framework, therefore, interprets consistency restoration as a controlled reconfiguration process operating over the retail configuration space.

5. Consistency Configuration Layer

The Consistency Configuration Layer represents the central operational and conceptual component of the proposed framework. This layer is responsible for transforming inconsistent retail operational states into appropriate retail representations using the configuration rules and transformation operators.

Unlike traditional data-cleaning approaches focused primarily on filtering corrupted records, the proposed layer explicitly models inconsistencies as invalid configuration states requiring data diagnostics and controlled repair processes. Consequently, the layer operates as a retail reconfiguration engine, preserving explainability and transformation traceability.

The primary role of the layer is to establish a consistency-preserving transformation mechanism capable of converting BRS into valid retail configurations. The layer performs four principal tasks, among which are the identification of violated constraints, the classification of inconsistency states, the selection of appropriate configuration operators, and the generation of consistent retail representations. The layer, therefore, bridges the gap between noisy operational retail environments and downstream optimization systems because it requires feasible and reliable input representations.

The first stage of the layer involves the diagnosis of inconsistent states within the retail space. Diagnosis mechanisms analyze retail entities and relationships against predefined consistency constraints. Given a retail state S, the diagnosis process evaluates:

$$D(S) = \{c_i \in C | c_i(S) = false\}$$

where:

- $D(S)$ denotes the set of violated constraints,
- C denotes the set of consistency constraints.

The diagnosis process, therefore, identifies structural inconsistencies, semantic contradictions, temporal conflicts, and operational violations. The framework assumes that not all inconsistencies originate from system-level failures. Constraint violations may also result from human activities, including manual shelf adjustments, incomplete replenishment execution, delayed reporting, incorrect data entry, or deviations from prescribed planograms. Consequently, diagnosis mechanisms may identify inconsistencies whose ultimate resolution requires human intervention, physical verification, or managerial decision-making. In such situations, the Consistency Configuration Layer functions as a decision-support mechanism rather than a fully autonomous repair system. Detected inconsistencies are subsequently classified into BRS categories. The framework assumes that multiple inconsistency types may occur simultaneously within a single retail state. Consequently, diagnosis procedures operate on interconnected constraint structures.

Following diagnosis, the layer applies configuration operators responsible for transforming invalid retail states into consistent representations. Configuration operators constitute controlled transformation mechanisms defined over the retail configuration space.

Let:

$$O = \{o_1, o_2, \dots, o_n\}$$

denote the set of available configuration operators.

Each operator:

$$o_1 : S \rightarrow S'$$

transforms a retail state S into a modified state S'.

The framework introduces several classes of configuration operators. They are particularly important in large-scale distributed retail architectures.

1. Completion operators introduce missing assignments required for structural consistency. Some examples include assigning missing shelf locations, restoring incomplete product metadata, and reconstructing missing planogram relationships.
2. Removal operators eliminate contradictory or invalid entities and relationships. They are represented by the removal of duplicate shelf assignments, the deletion of obsolete promotional records, and the elimination of invalid category mappings.
3. Substitution operators replace invalid entities or assignments with admissible alternatives. Common examples include relocation of products to compatible shelf zones, replacement of invalid replenishment paths, and reassignment of incompatible category placements.

4. Synchronization operators restore temporal consistency between distributed retail systems. Under their responsibility are the synchronization of inventory states, the update of delayed planograms, and the alignment of pricing records.
5. Consolidation operators merge conflicting representations originating from heterogeneous systems into unified retail states.

The proposed framework adopts a minimal-disruption transformation strategy. It seeks to minimize modifications required to restore consistency. The formulation of it preserves operational continuity and reduces unnecessary modifications to existing retail structures. Formally, the transformation objective may be expressed as:

$$\min \Delta(S, S')$$

subject to:

$$\forall c_i \in C: c_i(S') = true$$

where $\Delta(S, S')$ denotes an abstract transformation distance metric measuring the magnitude of modifications required to transform state S into state S'. Depending on implementation requirements, the distance function may incorporate weighted graph-edit distances over retail entity relationships, penalties associated with constraint violations, or vector-based similarity measures over retail attributes.

The proposed framework treats configuration operators as coordinated repairing mechanisms. Operator execution follows an iterative diagnosis–repair–validation cycle. After each operator application, the resulting retail state is re-evaluated against the complete set of consistency constraints. Consequently, newly introduced violations are detected during the subsequent diagnosis phase and become inputs to the next transformation iteration.

Consistency restoration is viewed as a sequence of controlled configuration transitions. If the repair of one inconsistency introduces another constraint violation (e.g., relocation of a product to resolve a temporal inconsistency that subsequently exceeds shelf capacity), the new violation is treated as a newly identified Broken Retail State and handled during the following iteration.

To reduce unnecessary oscillations, the framework conceptually assumes operator prioritization. Structural consistency is restored before semantic consistency, followed by temporal synchronization and operational refinement. This ordering reflects the dependency between retail constraints.

The transformation process terminates when either (i) all consistency constraints are satisfied, producing a Consistent Retail State, or (ii) no further operator can improve the current configuration according to the available repair knowledge. The latter situation represents a stable, unresolved configuration requiring external intervention or additional domain knowledge. Formal convergence guarantees and optimal repair strategies remain subjects of future research.

The proposed Broken Retail State concept may be viewed as a domain-specific realization of invalid configurations studied in knowledge-based configuration systems. Similarly, the Consistency Configuration Layer corresponds to a diagnosis-and-repair mechanism operating over constrained retail configuration spaces.

An important characteristic of the Consistency Configuration Layer is preservation of explainability throughout the transformation process. Each configuration operator is applied to a retail state and produces traceable transformation records. They are describing violated constraints, selected repair operators, affected entities, and resulting state modifications. This capability enables operational auditing, configuration explainability, debugging of retail systems, and analysis of recurring inconsistency patterns.

The explicit representation of inconsistencies additionally allows future integration with intelligent reasoning systems, AI-driven diagnosis mechanisms, and adaptive retail

reconfiguration approaches. The Consistency Configuration Layer models inconsistency restoration as a configuration process operating over constrained retail state spaces. So the proposed framework introduces a structured foundation for future research on intelligent retail reconfiguration systems.

The framework additionally enables future integration with constraint satisfaction methods, Answer Set Programming, ontology-based reasoning, explainable AI systems, adaptive optimization algorithms, and sustainability-aware retail analytics. Consequently, the proposed layer provides both a theoretical and operational basis for consistency-driven retail configuration research.

6. Illustrative Scenarios

To demonstrate the applicability of the Retail Configuration Framework, this section presents representative illustrative scenarios of possible BRS and their corresponding configuration transformations. The objective is to clarify how inconsistency states are modeled, diagnosed, and transformed within the Consistency Configuration Layer.

Each scenario follows a uniform structure: (i) initial broken state, (ii) violated constraints, and (iii) configuration-based repair leading to a consistent retail state.

1. *Scenario 1.* Shelf capacity overflow (structural BRS). When multiple replacement candidates exist, substitution operators may use priority scores derived from profitability, promotional importance, category coverage, or contractual obligations.

   Initial state. A set of products is assigned to a shelf exceeding its physical capacity due to independent updates from assortment planning and promotional expansion.
   Violation. Structural constraint violation (shelf capacity exceeded).

   $$\sum_{i=1}^{n} w_i x_i > W_s$$

   Configuration process. Removal operators eliminate low-priority promotional items. Substitution operators relocate oversized SKUs to alternative shelves. Consolidation operators reconcile duplicate allocations created by different systems.
   Resulting state. A feasible shelf configuration satisfying physical constraints while preserving high-priority product placement.

2. *Scenario 2.* Phantom inventory (semantic BRS). The acceptable synchronization latency depends on the operational context and may range from near real-time updates in smart retail environments to periodic synchronization in conventional retail systems.

   Initial state. Inventory systems report the availability of a product, but shelf observation data indicate the absence of physical stock.
   Violation. Semantic inconsistency between the inventory database and shelf state.
   Configuration process. Synchronization operators align inventory records with observed shelf data. Completion operators infer missing replenishment events. Removal operators discard stale inventory entries.
   Resulting state. A consistent mapping between digital inventory representation and physical shelf state.

3. *Scenario 3.* Duplicate shelf assignment (structural + operational BRS). Consolidation operators may rely on entity-resolution and record-deduplication mechanisms to determine authoritative representations among conflicting assignments.

Initial state. A single SKU is assigned to multiple incompatible shelf zones due to conflicting planogram updates.
Violation. Structural constraint violation (unique placement requirement). Operational constraint violation (invalid execution plan).
Configuration process. Consolidation operators merge conflicting assignments. Removal operators eliminate redundant placements. Substitution operators reassign SKUs based on priority rules.
Resulting state. Each SKU is assigned to a single valid shelf location consistent with operational constraints.

4. *Scenario 4.* Stale planogram configuration (temporal BRS). Depending on deployment architecture, synchronization and repair operators may operate either continuously in real time or as scheduled batch consistency-restoration processes.

   Initial state. A planogram references discontinued products and outdated shelf layouts.
   Violation. Temporal inconsistency (expired configuration reference). Structural inconsistency (invalid entity references).
   Configuration process. Synchronization operators update the planogram against the current product catalog. Removal operators eliminate obsolete product references. Completion operators reconstruct missing layout mappings.
   Resulting state. A temporally consistent planogram aligned with the current retail assortment.

5. *Scenario 5.* Promotion-placement conflict (Semantic + operational BRS). When multiple promotional placements satisfy the applicable constraints, operator selection may follow retailer-defined merchandising policies, including promotion priority, available promotional space, campaign objectives, and category-specific display rules.

   Initial state. A product is flagged as promotional in the system, but is not placed in designated promotional zones.
   Violation. Semantic inconsistency (promotion flag mismatch). Operational constraint violation (placement rule breach)
   Configuration process. Substitution operators relocate products to promotional shelf zones. Completion operators assign missing promotional attributes in shelf representation. Consolidation operators reconcile promotional metadata across systems.
   Resulting state. Consistent alignment between promotional status and physical shelf placement.

6. *Scenario 6.* Regional assortment violation (contextual BRS). Configuration operators may evaluate region-specific business rules, regulatory requirements, and localized assortment policies to ensure that the resulting configuration complies with the operational context of the target store.

   Initial state. A set of products appears in a store region where they are not authorized due to regional assortment policies.
   Violation. Contextual constraint violation (policy mismatch).
   Configuration process. Removal operators eliminate non-compliant products. Substitution operators replace them with regionally approved alternatives. Completion operators restore missing assortment coverage where necessary.
   Resulting state. A regionally compliant assortment configuration consistent with business policies.

7. Discussion

The proposed framework introduces a conceptual shift in the treatment of retail data inconsistencies by interpreting them as configuration violations rather than conventional data quality issues. This reinterpretation enables the formal modeling of inconsistent retail states as BRS within a structured configuration space. As a result, inconsistencies are transformed into valid configurations through configuration operators.

A key contribution of this approach lies in the separation between inconsistency management and optimization. Traditional retail optimization methods typically assume the existence of clean and reliable input data. However, real-world retail environments frequently violate this assumption due to distributed data sources, asynchronous updates, and operational complexity. The proposed framework addresses this gap by introducing a dedicated consistency configuration layer that restores structural validity prior to optimization.

The proposed framework differs from existing retail optimization approaches in several important ways. Most SSAP and retail planning models assume that input data is already valid and focus on optimizing shelf allocation, assortment planning, or replenishment decisions [2–6]. In contrast, the proposed framework addresses the preceding problem of consistency restoration and explicitly models invalid retail representations as Broken Retail States. The framework also differs from traditional configuration research, which primarily investigates product, software, or feature-model configuration [7–9]. Instead, it extends configuration concepts to operational retail data and introduces a dedicated consistency-configuration layer responsible for transforming inconsistent retail states into admissible retail configurations. To the best of our knowledge, this perspective has not been formalized previously within retail shelf management and retail configuration research.

Another important aspect of the framework is its emphasis on explainability and traceability. Each transformation applied within the configuration layer is explicitly associated with violated constraints and selected repair operators. Because of this, the full auditability of the consistency restoration process becomes possible. It is particularly relevant in large-scale retail systems where multiple stakeholders and automated processes interact simultaneously.

The operator-based formulation further enhances the flexibility of the framework. The modular configuration operators support a wide range of inconsistency types without requiring domain-specific reengineering. Common operators are completion, removal, substitution, synchronization, and consolidation of the system.

From a theoretical perspective, the framework extends configuration research by broadening the notion of configurability beyond product or system design to include operational data states themselves. In this view, inconsistency becomes a property of configuration spaces. This perspective aligns retail data management with broader research in configuration systems, reconfiguration processes, and knowledge-based reasoning.

The designed framework provides a structured, configuration-theoretic interpretation of retail data inconsistencies and positions them as first-class objects within retail system design and analysis.

An important limitation of purely technical consistency restoration approaches is that retail environments are inherently socio-technical systems. Retail representations are created and maintained by multiple actors operating at different organizational levels, including planners, category managers, store employees, warehouse personnel, and logistics operators. These stakeholders often possess different objectives, responsibilities, and levels of expertise, which may contribute to the emergence of inconsistent retail states. Furthermore, certain inconsistencies originate from physical events such as theft, product damage, misplaced inventory, or losses during transportation. Such situations cannot always be resolved solely through automated configuration operators. Consequently, the proposed framework should be

interpreted as a human-in-the-loop consistency management approach, where automated diagnosis and repair mechanisms complement rather than replace human judgment and physical verification processes.

8. Conclusion and Future Work

This paper introduces a configuration-oriented framework for configuring and repairing inconsistent retail data. The operational anomalies within a structured retail configuration space are referred to as BRS. Unlike traditional approaches that treat inconsistencies as data quality defects to be filtered or corrected in an ad hoc manner, the proposed framework formalizes inconsistency as a concept in configuration systems. This shift enables a systematic treatment of retail data irregularities as invalid configurations that can be diagnosed, transformed, and restored through configuration operators.

The central contribution of the work is the Consistency Configuration Layer, which separates raw retail data from downstream optimization processes and provides a dedicated mechanism for consistency restoration. Within this layer, inconsistent retail states are identified through constraint violation analysis, classified into structural, semantic, temporal, operational, and contextual categories, and subsequently transformed into valid retail configurations. The transformation process is governed by a set of modular configuration operators, including completion, removal, substitution, synchronization, and consolidation, which collectively ensure traceable and explainable state repair.

This research introduces a formal model of BRS and defines a corresponding configuration space. Therefore, a theoretical foundation for treating retail inconsistencies as structured objects is established. This approach allows retail systems to separate inconsistency handling and optimization, ensuring that downstream analytical methods operate exclusively on valid and coherent data.

The proposed framework contributes to ongoing research on retail planning and configuration-based reasoning by extending consistency management concepts beyond traditional optimization and configuration domains [1,7,9]. It highlights the importance of consistency restoration as a prerequisite for reliable decision-making in large-scale and data-intensive retail environments.

The primary limitation of the present work is its conceptual nature. The framework has been developed as a theoretical and architectural model and has been illustrated through representative retail scenarios rather than evaluated using real-world retail datasets. Consequently, the effectiveness, scalability, and practical impact of the proposed consistency restoration mechanisms remain subjects for future empirical investigation.

Despite its conceptual contributions, several directions remain open for future research. First, the formalization of optimal repair strategies within the Consistency Configuration Layer could be further developed, particularly with respect to minimal-change principles and cost-aware transformations. Second, the integration of uncertainty modeling would enhance the framework's ability to handle ambiguous or partially observed retail states, potentially through probabilistic configuration models or fuzzy constraint representations. Third, the incorporation of automated reasoning techniques such as Constraint Satisfaction Problems, Answer Set Programming, and SAT-based solvers could provide a more rigorous computational foundation for inconsistency diagnosis and repair.

A particularly important direction for future research is the empirical validation of the proposed framework using operational retail data obtained from real retail environments. Such studies could evaluate the frequency of Broken Retail States, the effectiveness of different configuration operators, the quality of generated repairs, and the impact of consistency restoration on downstream optimization performance. Empirical validation would provide

important evidence regarding the practical applicability and scalability of the proposed approach.

Additionally, future work may explore the integration of machine learning methods for predicting inconsistency patterns and guiding configuration operator selection. Hybrid approaches combining symbolic reasoning with data-driven models may further improve scalability and adaptability in highly dynamic retail environments. Finally, empirical validation of the framework using real-world retail datasets would be a valuable extension, enabling quantitative evaluation of its effectiveness in improving consistency, traceability, and downstream optimization quality.

Future research should also investigate human-centered aspects of retail consistency management. In particular, the interaction between retail personnel and consistency restoration systems deserves further attention. User roles, interface design, workflow integration, explainability of repair recommendations, and human acceptance of automatically generated configuration changes may substantially influence the effectiveness of consistency restoration processes. Incorporating human-in-the-loop validation mechanisms represents a promising direction for extending the proposed framework.

In conclusion, the proposed framework establishes a novel configuration-theoretic perspective on retail data inconsistencies. It provides a structured foundation for research on consistency-aware retail systems, intelligent reconfiguration mechanisms, and AI-supported decision-making in complex operational environments.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT (OpenAI GPT-4) in order to improve language clarity and consistency, refine grammar and spelling, and assist in restructuring text for readability. The scientific content, core ideas, methodology, analysis, and conclusions were developed by the author(s). After using this tool, the author(s) carefully reviewed and edited the output and take full responsibility for the final content of the publication.

References

[1] T. Bianchi-Aguiar, A. Hübner, M.A. Carravilla, J.F. Oliveira, Retail shelf space planning problems: A comprehensive review and classification framework, Eur. J. Oper. Res. 289 (2021) 1–16. https://doi.org/10.1016/j.ejor.2020.06.018.

[2] A. Hübner, T. Düsterhöft, M. Ostermeier, Shelf space dimensioning and product allocation in retail stores, Eur. J. Oper. Res. 292 (2021) 155–171. https://doi.org/10.1016/j.ejor.2020.10.030.

[3] A. Hübner, H. Kuhn, Decision support for managing assortments, shelf space, and replenishment in retail, Flex. Serv. Manuf. J. 36 (2024) 1–35. https://doi.org/10.1007/s10696-023-09492-z.

[4] A. Hübner, K. Schaal, Effect of replenishment and backroom on retail shelf-space planning, Bus. Res. 10 (2017) 123–156. https://doi.org/10.1007/s40685-016-0043-6.

[5] T. Flamand, A. Ghoniem, B. Maddah, Promoting impulse buying by allocating retail shelf space to grouped product categories, J. Oper. Res. Soc. 67 (2016) 953–969. https://doi.org/10.1057/jors.2015.120.

[6] M.H. Yang, W.C. Chen, Study on shelf space allocation and management, Int. J. Prod. Econ.

60 (1999) 309–317. https://doi.org/10.1016/S0925-5273(98)00134-0.

[7] A. Felfernig, S. Reiterer, M. Stettinger, F. Reinfrank, M. Jeran, G. Ninaus, Recommender systems for configuration knowledge engineering, in: CEUR Workshop Proc., 2013: pp. 51–54.

[8] A. Felfernig, M. Schubert, C. Zehentner, An efficient diagnosis algorithm for inconsistent constraint sets, Artif. Intell. Eng. Des. Anal. Manuf. AIEDAM 26 (2012). https://doi.org/10.1017/S0890060411000011.

[9] J.A. Pereira, M. Acher, H. Martin, J.M. Jézéquel, G. Botterweck, A. Ventresque, Learning software configuration spaces: A systematic literature review, J. Syst. Softw. 182 (2021). https://doi.org/10.1016/j.jss.2021.111044.

[10] K. Schaal, A. Hübner, When does cross-space elasticity matter in shelf-space planning? A decision analytics approach, Omega (United Kingdom) 80 (2018) 135–152. https://doi.org/10.1016/j.omega.2017.08.015.

[11] A. Hübner, A decision support system for retail assortment planning, Int. J. Retail Distrib. Manag. 45 (2017) 808–825. https://doi.org/10.1108/IJRDM-09-2016-0166.

[12] K. Czerniachowska, M. Hernes, A heuristic approach to shelf space allocation decision support including facings, capping, and nesting, Symmetry (Basel). 13 (2021) 1–18. https://doi.org/10.3390/sym13020314.

[13] K. Czerniachowska, K. Michalak, M. Hernes, Heuristics for the shelf space allocation problem, OPSEARCH 60 (2023) 835–869. https://doi.org/10.1007/s12597-023-00636-1.

[14] K. Czerniachowska, M. Hernes, A genetic algorithm for the shelf-space allocation problem with vertical position effects, Mathematics 8 (2020) 1–20. https://doi.org/10.3390/math8111881.

[15] K. Czerniachowska, M. Hernes, Simulated annealing hyper-heuristic for a shelf space allocation on symmetrical planograms problem, Symmetry (Basel). 13 (2021) 1182. https://doi.org/10.3390/sym13071182.

[16] N. Borin, P.W. Farris, J.R. Freeland, A Model for Determining Retail Product Category Assortment and Shelf Space Allocation, Decis. Sci. 25 (1994) 359–384. https://doi.org/10.1111/j.1540-5915.1994.tb01848.x.

[17] S. Chadha, Vision-Based Object Recognition in Retail, 2025.

[18] R.P. Mesquita, F. Leal, J.A. De Queiroz, Digital Twins in The Retail Industry: A Systematic Literature Review, Int. J. Simul. Model. 23 (2024). https://doi.org/10.2507/IJSIMM23-3-690.

[19] V.F. Yu, R. Maglasang, Y.C. Tsao, Shelf space allocation problem under carbon tax and emission trading policies, J. Clean. Prod. 196 (2018) 438–451. https://doi.org/10.1016/j.jclepro.2018.05.165.

[20] K. Czerniachowska, S. Subbotin, Merchandising rules for shelf space allocation with product categorization and vertical positioning, Inform. Ekon. 2021 (2021) 34–59. https://doi.org/10.15611/ie.2021.1.02.

[21] K. Czerniachowska, K. Lutosławski, M. Hernes, Linear and nonlinear shelf space allocation

problems with vertical and horizontal bands, J. Econ. Manag. 44 (2022) 119–141. https://doi.org/10.22367/jem.2022.44.06.

[22] K. Czerniachowska, M. Hernes, K. Lutosławski, A linearisation approach to solving a non-linear shelf space allocation problem with multi-oriented capping in retail store and distribution centre, Oper. Res. Decis. 32 (2022). https://doi.org/10.37190/ord220403.

[23] K. Czerniachowska, M. Hernes, Shelf Space Allocation for Specific Products on Shelves Selected in Advance, Eur. Res. Stud. J. XXIV (2021) 316–334. https://doi.org/10.35808/ersj/2356.

[24] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Consistency-based diagnosis of configuration knowledge bases, Artif. Intell. 152 (2004). https://doi.org/10.1016/S0004-3702(03)00117-6.

[25] P. Lambrix, Completing and Debugging Ontologies: State-of-the-art and Challenges in Repairing Ontologies, J. Data Inf. Qual. 15 (2023). https://doi.org/10.1145/3597304.

[26] S.N. Jayakrishnan, Artificial Intelligence (AI) in Retailing- A Systematic Review and Research Agenda, SSRN Electron. J. (2022). https://doi.org/10.2139/ssrn.4134959.

[27] Y. Nomura, Z. Liu, T. Nishi, Deep Reinforcement Learning for Dynamic Pricing and Ordering Policies in Perishable Inventory Management, (2025). https://doi.org/10.3390/app15052421.

[28] S.C. Wu, W.Y. Chiu, C.F. Wu, Deep Reinforcement Learning for Task Assignment and Shelf Reallocation in Smart Warehouses, IEEE Access 12 (2024). https://doi.org/10.1109/ACCESS.2024.3392752.

[29] K. Shchekotykhin, G. Friedrich, P. Fleiss, P. Rodler, Interactive ontology debugging: Two query strategies for efficient fault localization, J. Web Semant. 12–13 (2012). https://doi.org/10.1016/j.websem.2011.12.006.

[30] B. Kimelfeld, E. Livshits, L. Peterfreund, Counting and enumerating preferred database repairs, Theor. Comput. Sci. 837 (2020). https://doi.org/10.1016/j.tcs.2020.05.016.

[31] A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E.C. Teppan, Industrial Applications of Answer Set Programming, KI - Kunstl. Intelligenz 32 (2018) 165–176. https://doi.org/10.1007/s13218-018-0548-6.

[32] R. Kaminski, T. Schaub, On the Foundations of Grounding in Answer Set Programming, Theory Pract. Log. Program. 23 (2023). https://doi.org/10.1017/S1471068422000308.

[33] R. Gonçalves, M. Knorr, J. Leite, Forgetting in Answer Set Programming-A Survey, Theory Pract. Log. Program. 23 (2023). https://doi.org/10.1017/S1471068421000570.

[34] C. V. Trappey, A.J.C. Trappey, A chain store marketing information system: Realizing Internet-based enterprise integration and electronic commerce, Ind. Manag. Data Syst. 98 (1998). https://doi.org/10.1108/02635579810227733.

[35] A.H. Hübner, H. Kuhn, Retail category management: State-of-the-art review of quantitative research and software applications in assortment and shelf space management, Omega 40 (2012) 199–209. https://doi.org/10.1016/j.omega.2011.05.008.

[36] A. Hübner, K. Schaal, An integrated assortment and shelf-space optimization model with demand substitution and space-elasticity effects, Eur. J. Oper. Res. 261 (2017) 302–316.

https://doi.org/10.1016/j.ejor.2017.01.039.

[37] M.R. Kuiti, D. Ghosh, S. Gouda, S. Swami, R. Shankar, Integrated product design, shelf-space allocation and transportation decisions in green supply chains, Int. J. Prod. Res. 57 (2019) 6181–6201. https://doi.org/10.1080/00207543.2019.1597292.

[38] T. Wulfert, R. Schütte, Retailer's Dual Role in Digital Marketplaces: Reference Architectures for Retail Information Systems, SN Comput. Sci. 3 (2022). https://doi.org/10.1007/s42979-022-01098-w.

[39] Y. Chen, Z. Wu, Y. Wang, Omnichannel product selection and shelf space planning optimization, Omega 127 (2024) 103074. https://doi.org/https://doi.org/10.1016/j.omega.2024.103074.

Learning to Solve and Optimize by Evolving Code*

Veronika Semmelrock[1,†], Benedetta Strizzolo[2,†], Francesco Zuccato[1,†], Gerhard Friedrich[1,‡], Patrick Rodler[1,‡] and Konstantin Schekotihin[1,‡]

[1]University of Klagenfurt, Klagenfurt, Austria
[2]University of Udine, Udine, Italy

Abstract

Combinatorial and optimization problems are central to many industrial AI applications, including automated engineering, product configuration, and production scheduling. A long-standing vision in AI is that domain experts should specify *what* constitutes a correct solution, while the system determines *how* to find one efficiently. In practice, however, deploying state-of-the-art solvers on large real-world instances still demands substantial expert intervention: redesigning problem encodings, crafting domain-specific heuristics, decomposing problems, or implementing tailored local-search procedures.

To address these challenges, we build on recent advances in program generation via code evolution [1]. OpenEvolve [2], an open-source implementation of [1], consists of four modules: a *prompt sampler*, an *LLM ensemble*, an *evaluator pool*, and a *program database*. We introduce CheckMate [3], which extends the *evaluator pool*. OpenEvolve+CheckMate only requires as inputs *what* a solution is, via *(i)* a formal specification F of valid solutions, *(ii)* a natural language description D of the problem, and *(iii)* a small set of training problem instances I.

OpenEvolve+CheckMate outputs the best Python program p^* from an evolutionary training loop. In each iteration j, the LLM ensemble generates a program p_j, which CheckMate evaluates by *(a)* executing p_j on each $i \in I$, producing a candidate solution c_{ji}, *(b)* syntactically validating c_{ji}, *(c)* verifying its correctness against F given i using a state-of-the-art solver (such as clingo [4]), and *(d)* scoring p_{ji} depending on its performance. CheckMate returns the score of p_j and, for incorrect outputs, textual feedback per incorrect c_{ji}. Finally, the best Python program p^* corresponds to the highest-scored p_j after a predefined number of iterations.

We investigate: (RQ1) *Can OpenEvolve+CheckMate solve hard real-world* (a) *combinatorial and* (b) *optimization problems, such as configuration and scheduling?*, (RQ2) *How does the performance of the generated programs compare to state-of-the-art solvers?*, and (RQ3) *How well do the evolved programs scale?*.

We evaluate on three industrial case studies: two (decisional) configuration problems, the House Configuration Problem (HCP) [5, 6] and the Combined Configuration Problem (CCP) [7], and one (optimization) scheduling problem, the Energy-Aware Double-Flexible Job-Shop Problem (E-DFJSP) [8], which lexicographically minimizes tardiness, energy consumption, and makespan. As baselines we use clingo [4] for HCP and CCP and CP Optimizer [9] for E-DFJSP, each using the original formal problem encodings [10] [11] [12] during testing. Test sets over-represent hard instances (RQ3). LLM program generation queries were served by GPT-5 (60%) and GPT-5-mini (40%). Overall training cost for all experiments was below €200 in API fees, with training time between 1-16 hours per evolution run, depending on the problem.

In all case studies, the evolved programs outperform the baselines. On HCP, the evolved program solves all instances (avg. 0.08 *s*, 281 MB), while clingo fails all hard instances. On CCP, the evolved program solves 89% of instances, and a portfolio of four independently evolved programs solves 100%—*to our knowledge, for the first time solving all CCP instances from [7].* Clingo solves 65% overall and 4% of hard instances. On E-DFJSP, the evolved program solves 100% of instances, each within 4 *s*, while CP Optimizer fails on 14 of the 18 hardest instances and, on instances with 200+ jobs, is outperformed on tardiness—the highest-priority objective—by 49% on average.

These results show that OpenEvolve+CheckMate needs no information on *how* to construct solutions, only *what* defines a correct (optimal) solution and a set of representative instances, and that the resulting programs can efficiently solve hard real-world *(a)* combinatorial and *(b)* optimization problem instances currently out of reach for state-of-the-art solvers—addressing (RQ1–RQ3). Future work topics include ablation studies of CheckMate's components, comparisons with open-weight LLMs, and analyses of CheckMate's adoptability in dynamic environments with changing problem constraints.

Acknowledgments

This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/COE12 and the Austrian Research Promotion Agency (FFG) FO999910235 (SAELING) and 930480ATRIA (ATRIA).

References

[1] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. Mehrabian, et al., Alphaevolve: A coding agent for scientific and algorithmic discovery, arXiv preprint arXiv:2506.13131 (2025).

[2] A. Sharma, Openevolve: an open-source evolutionary coding agent, https://github.com/algorithmicsuperintelligence/openevolve, 2025. Accessed: 14 April 2026.

[3] V. Semmelrock, B. Strizzolo, F. Zuccato, G. Friedrich, P. Rodler, K. Schekotihin, Learning to solve and optimize by evolving code, in: IJCAI, 2026. To appear.

[4] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, P. Wanko, Potassco guide version 2.2.0, 2019. URL: https://github.com/potassco/guide/releases/tag/v2.2.0, retrieved from https://github.com/potassco/guide/releases/tag/v2.2.0.

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

[6] G. Friedrich, A. Ryabokon, A. A. Falkner, A. Haselböck, G. Schenner, H. Schreiner, (re)configuration based on model generation, in: C. Drescher, I. Lynce, R. Treinen (Eds.), LoCoCo 2011, volume 65 of *EPTCS*, 2011, pp. 26–35. doi:10.4204/EPTCS.65.3.

[7] M. Gebser, A. S. Ryabokon, G. Schenner, Solving combined configuration problems: a heuristic approach, in: Configuration Workshop, 2015. URL: https://api.semanticscholar.org/CorpusID:11120414.

[8] G. Gong, Q. Deng, X. Gong, W. Liu, Q. Ren, A new double flexible job-shop scheduling problem integrating processing time, green production, and human factor indicators, Journal of Cleaner Production 174 (2018) 560–576.

[9] P. Laborie, J. Rogerie, P. Shaw, P. Vilím, IBM ILOG CP Optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG, Constraints 23 (2018) 210–250.

[10] V. Semmelrock, G. Friedrich, Investigating the grounding bottleneck for a large-scale configuration problem: Existing tools and constraint-aware guessing, EPTCS 439 (2026) 482–495. URL: http://dx.doi.org/10.4204/EPTCS.439.33. doi:10.4204/eptcs.439.33.

[11] M. Gebser, M. Maratea, F. Ricca, The sixth answer set programming competition, JAIR 60 (2017) 41–95.

[12] F. Zuccato, P. Rodler, G. Friedrich, K. Schekotihin, R. Comploi-Taupe, Energy-aware double-flexible job shop scheduling with machine modes and setup times: A real-world industrial case study using constraint programming, in: ECAI Workshop on AI-based Planning for Complex Real-World Applications (CAIPI'25), volume 4103 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2025, pp. 84–99. URL: https://ceur-ws.org/Vol-4103/.

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany

*This is an extended abstract summarizing the full paper published at the 31st International Joint Conference on Artificial Intelligence (IJCAI 2026).

†The first three authors contributed equally and are listed in alphabetical order.

‡The last three authors contributed equally and are listed in alphabetical order.

veronika.semmelrock@aau.at (V. Semmelrock); strizzolo.benedetta@spes.uniud.it (B. Strizzolo); francesco.zuccato@aau.at (F. Zuccato); gerhard.friedrich@aau.at (G. Friedrich); patrick.rodler@aau.at (P. Rodler); konstantin.schekotihin@aau.at (K. Schekotihin)

0009-0005-8435-1186 (V. Semmelrock); 0009-0004-1985-6914 (F. Zuccato); 0000-0002-1992-4049 (G. Friedrich); 0000-0001-8178-4692 (P. Rodler); 0000-0002-0286-0958 (K. Schekotihin)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Reconciling Product Flexibility with Cost, Delivery, and Quality: The Importance of Bundling Online Sales Configurator, Product Modularity, and Knowledge Absorption from Customers

Alessio Trentin[1], Enrico Sandrin[1,*], Svetlana Suzic[1], Chiara Grosso[2] and Cipriano Forza[1]

[1] *University of Padova, Stradella San Nicola 3, 36100 Vicenza, Italy*
[2] *Sapienza University of Rome, Piazzale Aldo Moro 5, 00185 Rome, Italy*

Extended abstract of Trentin et al. [1]

In response to recent calls to intensify research on the relationships among the enablers of mass customization capability (MCC) as well as on their combined effects on this organizational capability, this study is the first to investigate the combined effect of three practices that correspond one-to-one with Salvador et al.'s [2] three fundamental building blocks of mass customization: knowledge absorption from customers (KAfC), corresponding to solution space development; product modularity (PM), corresponding to robust process design; and online sales configurator use (OSCU), corresponding to choice navigation. As a matter of fact, Salvador et al. [2] had not clarified whether these building blocks should be developed simultaneously or sequentially. This is also the first study focused on MCC to utilize concepts from resource orchestration theory, a theory highly relevant for understanding how organizational capabilities are generated but not previously applied to MCC antecedents research.

The research hypothesis is based on Salvador et al.'s [2] conceptual model, combined both with the principle of resource orchestration theory that posits that organizational capabilities arise from the combination of multiple interdependent resources that reinforce one another (resource bundle), and with Venkatraman's [3] work on how to define and test the concept of fit. This high-level line of reasoning is supported by detailed arguments systematically explaining why each of the three practices (KAfC, PM, OSCU) facilitates the implementation of the other two and reinforces their benefits in terms of MCC.

The hypothesis is tested using covariance-based structural equation modeling (CB-SEM) on data from 213 manufacturing plants across three industries (electronics, machinery, and transportation equipment) and 16 countries in Asia, America, and Europe. Results strongly support the hypothesis. The joint effect of KAfC, PM, and OSCU explains 41.9% of MCC variance, compared to only 13.9% explained by their independent effects. This stronger explanatory power, greater than that of most previous studies on MCC predictors, challenges the essentially sequential logic that dominates the literature on the guidelines for the implementation of mass customization (see Suzić et al. [4]), suggesting the adoption of a holistic approach in which the three practices are implemented together.

Furthermore, since the three practices correspond to Salvador et al.'s [2] building blocks of mass customization, these results provide an initial answer, more than fifteen years later, to the question posed by Salvador et al. [2] regarding whether the three building blocks should be developed together or in a specific sequence. A further contribution consists in clarifying, by resolving apparent contradictions in the literature, the complex link between mass customization and flexibility. Finally, this study contributes to resource orchestration research by arguing that fit-as-covariation is most appropriate for modeling bundles of three or more interdependent, mutually reinforcing resources.

Keywords

Product configuration, Mass customization, Knowledge absorption from customers, Product modularity, Resource orchestration, Flexibility

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany
* Corresponding author.
alessio.trentin@unipd.it (A. Trentin); enrico.sandrin@unipd.it (E. Sandrin); svetlana.suzic.1@unipd.it (S. Suzic); chiara.grosso@uniroma1.it (C. Grosso); cipriano.forza@unipd.it (C. Forza)
0000-0002-7853-4104 (A. Trentin); 0000-0001-9170-0683 (E. Sandrin); 0000-0002-7969-875X (S. Suzic); 0009-0008-2691-3947 (C. Grosso); 0000-0003-4583-2962 (C. Forza)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

References

[1] A. Trentin, E. Sandrin, S. Suzic, C. Grosso, C. Forza, Reconciling product flexibility with cost, delivery, and quality: the importance of bundling mass customization practices, Global Journal of Flexible Systems Management 26 (2025) 269-293. doi: 10.1007/s40171-024-00429-5.

[2] F. Salvador, P. M. de Holan, F. Piller, Cracking the code of mass customization, MIT Sloan Management Review 50 (2009) 71-78.

[3] N. Venkatraman, The concept of fit in strategy research: toward verbal and statistical correspondence, Academy of Management Review 14 (1989) 423-444. doi: 10.5465/amr.1989.4279078.

[4] N. Suzić, C. Forza, A. Trentin, Z. Anišić, Implementation guidelines for mass customization: current characteristics and suggestions for improvement, Production Planning & Control 29 (2018) 856-871. doi: 10.1080/09537287.2018.1485983.

Explaining Product Configurator Usage in Engineering Companies: A Technology Acceptance Perspective*

Qingsong Zhao[1,*], Oliver Bleisinger[2], Lars Hvam[1] and Lars Lundberg Kristensen[3]

[1] *Technical University of Denmark, Lyngby, Denmark*
[2] *University of Kaiserslautern-Landau, Kaiserslautern, Germany*
[3] *NNE, Kalundborg, Denmark*

Abstract

Product configurators are widely used to automate design and quotation processes for customized products. However, in practice, not all product configurator projects are successfully implemented or adopted, especially in complex engineering companies. Despite the technical and financial challenges involved in developing configurators, many companies fail to achieve actual adoption of these IT tools. This issue is closely related to the acceptance and use of configurators by people. To better understand this issue, we apply the Technology Acceptance Model (TAM) as a lens to explain the reasons behind configurator adoption and non-adoption. Based on a single in-depth case in a pharma engineering company, informed by our research team's experience from configurator projects in other engineering sectors, we propose a context-specific extension of TAM for product configurators in engineering companies. The model is presented as a conceptual contribution rather than an empirically validated instrument, and we outline a roadmap for its validation. This study contributes to product configurator research by linking configurator development with technology acceptance theory, and provides practical guidance for developing configurators more effectively from the user perspective.

Keywords

Product Configurator, Engineering, Technology Acceptance Model

1. Introduction

How can a company design a large number of product variants while still maintaining production quality and efficiency? This is the case for Volvo Trucks, a market leader in heavy-duty trucks [1]. During an assembly line visit as part of a Product Platform course at Jönköping University, it was stated that “every single truck we manufacture can be different.” [2]. Managing such a high level of product variety is practically impossible through manual work alone.

The answer lies in the use of information technologies (IT), such as product configurators. Together with modular product architecture, product configurators enable companies to handle large numbers of variants efficiently and form a key foundation for the mass customization strategy [3], [4], [5]. This strategy is often described as the ability to provide tailored products through flexible processes and organizational structures while maintaining the low costs associated with standardized mass production [6].

*ConfWS'26: 28th International Workshop on Configuration, Aug 15--16, 2026, Bremen, Germany
1* Corresponding author.
qzha@dtu.dk (Q. Zhao); bleising@rptu.de (O. Bleisinger); lahv@dtu.dk (L. Hvam); LZKR@nne.com (L. L. Kristensen)
0009-0001-4725-5460 (Q. Zhao); 0009-0006-6844-9973 (O. Bleisinger); 0000-0002-7617-2971 (L. Hvam)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Product configurators are widely used for customized products. For example, customers can use configurators to design a unique Volvo truck, receive a quotation for the order, and enable the manufacturing site to obtain detailed bills of materials for production [7], as shown in Figure 1. The use of product configurators brings significant benefits, such as reduced lead time and improved quality [3]. Beyond the automotive industry, the use and benefits of product configurators in engineering companies have also been documented in several studies, where many positive effects have been observed [8], [9].

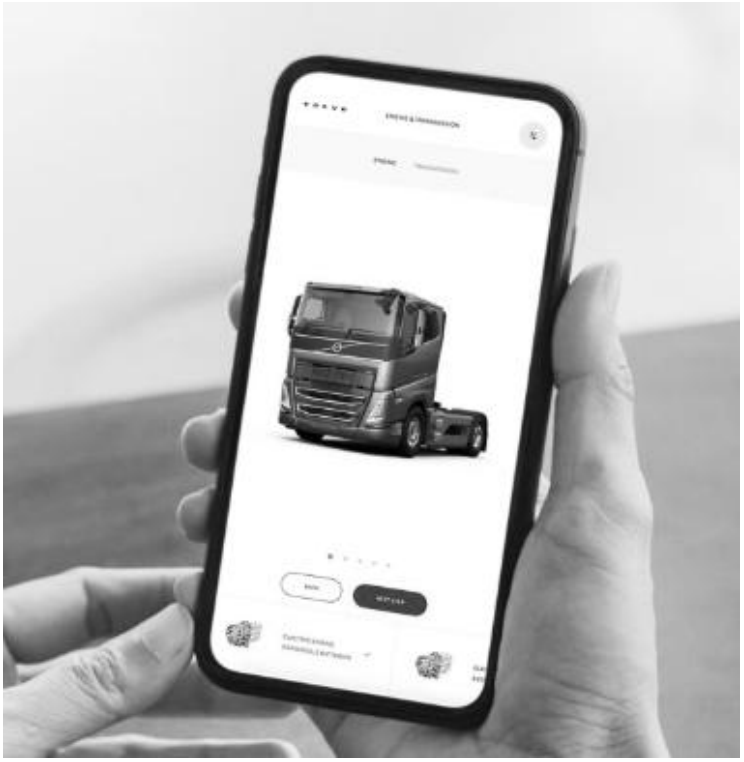


Figure 1: Design your Volvo truck [7].

Existing literature mainly reports successful cases and focuses on development or implementation procedures, frameworks, and strategies. However, even though product configurators are a mature technology, many configurator projects still fail in practice and there is much less attention to this problem in literature [10]. In some cases, the problem is not the technical capability of the configurator but the lack of consideration of organizational and human factors [11].

Product configurators are developed to support users, and regardless of how advanced a system is, it will not be successful if users are not properly involved or considered in its development and use. There is a need to better understand how users perceive and adopt product configurators.

To address this issue, theories from information systems research that focus on technology acceptance can provide useful insights. One widely used theory is the TAM, which explains technology adoption based on users' perceptions of usefulness and ease of use [12]. TAM has been applied to study the acceptance of various IT systems, but its application in the context of product configurators in complex engineering remains limited. Therefore, we propose a context-specific extension of TAM for product configurators in engineering companies, which provides an explanatory lens for understanding how users perceive, accept, and use product configurators. This model is grounded in a single in-depth industrial case and informed by broader cross-sector project experience; it is intended as an explanatory lens and a basis for future empirical validation, with the wider aim of drawing research attention to the human and organizational side of configurator adoption.

This paper is structured as follows. Section II reviews related literature. Section III describes the methodology. Section IV presents the model. Section V discusses the results. Section VI concludes the paper.

2. Related Literature

2.1. Product configurator

In general, product configurators are used to support the configuration process, from collecting customer needs to releasing the product documentation required for production [13]. These systems act as expert systems that help users create product specifications during the configuration process [14]. In engineering companies, existing literature documents numerous configurator implementations and highlights their potential to streamline processes and achieve various benefits [3], [9], [15], [16]. An example of a product configurator in an engineering company is illustrated by Shafiee in Figure 2.

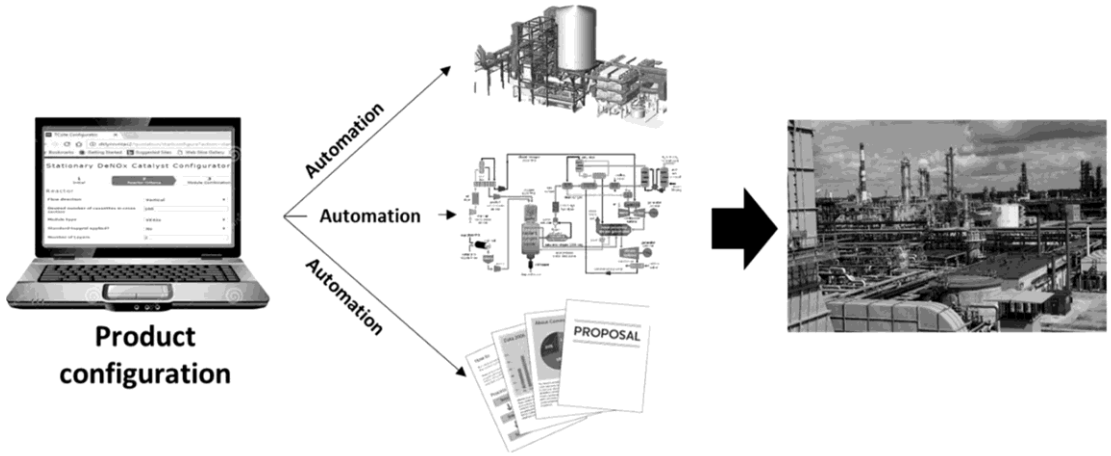


Figure 2: Product configurator in an engineering company [17].

Many studies have proposed frameworks for developing product configurators in engineering contexts. Hvam et al. present a seven-phase framework that includes analysis and redesign of specification processes, product modeling, software selection, and configurator modeling, implementation, and maintenance [3]. Zhao et al. emphasize the importance of the investigation phase and the alignment between product architecture and product configurators [18], [19]. Shafiee et al. present a framework for scoping product configurators, including identifying users and requirements, defining inputs and outputs, determining functionalities, and integrating product knowledge [20]. Kristjansdottir et al. propose a framework for identifying potential applications of product configurators, which includes identifying potential applications, aligning IT development, and establishing an overview of product configurator applications [21].

Besides many mature and established frameworks, several studies show that product configurator projects often fail not because of technical functionality. A recurring issue is that configurator development focuses mainly on software and product knowledge, while organizational and human aspects receive less attention [11]. In some cases, configurator projects also fail due to tacit knowledge, while the underlying issues are related to unclear concepts, poorly structured product knowledge, and difficulties in transferring knowledge from experts to product configurators [22]. Another common challenge is that configurators are developed based on assumptions about how people should work, rather than how configuration tasks are actually performed in practice [23].

Building on this, Haug et al. [10] study failed product configurator projects and identify five common causes of configurator failures, such as insufficient user involvement, weak alignment with organizational needs, lack of clarity about the purpose and scope of the configurator, as illustrated in Figure 3.

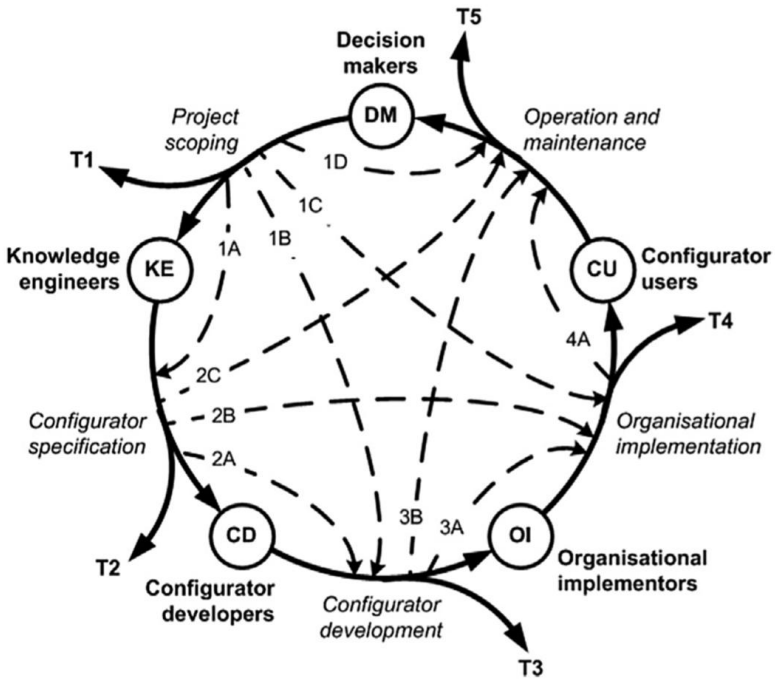


Figure 3: Product configurator project termination points [10]

While many frameworks provide strong guidance for developing and scoping product configurators in engineering companies, they mainly focus on technical and process-related aspects, and many configurator projects fail due to issues related to user understanding, involvement, and use in daily work. To better understand these issues, it is important to focus on how users perceive and adopt configurators.

2.2. TAM

TAM is a well-known model used to explain how users accept and use information technologies. TAM focuses on users rather than the technology itself, and it was first proposed by Davis with studies of acceptance of text editor application [12]. The model explains adoption mainly through two factors: perceived usefulness and perceived ease of use. Davis defined perceived usefulness as the degree to which the person believes that using the particular system would enhance her/his job performance, whereas the perceived ease of use was defined as the degree to which the person believes that using the particular system would be free of effort. These two factors influence users' attitude to use a system and, eventually, its actual use [12]. Meanwhile, these two factors are impacted by some external factors, as shown in Figure 4.

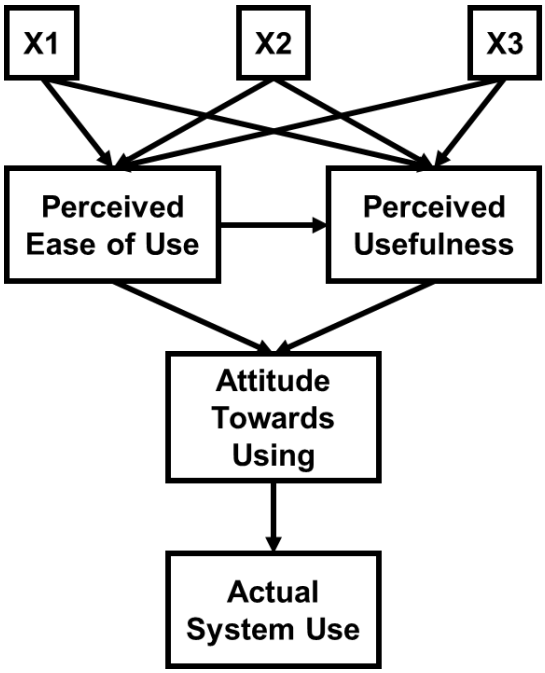


Figure 4: Technology acceptance model (adapted from [12])

Different versions and extensions of TAM have later been developed for different purposes, levels of detail, and application contexts [24], [25]. Nevertheless, TAM has been widely applied to study user acceptance of different software categories, such as office automation, software development and business application tools [26]. Numerous studies have shown that TAM is a useful theoretical model for understanding and explaining user behavior in IT implementation [27]. It has been tested in many empirical studies, and the measurement instruments used in TAM have been shown to be reliable and to produce consistent and valid results [26].

More recently, TAM has also been applied to newer digital technologies, such as AI-based technologies. Chatterjee et al. confirm that perceived usefulness and perceived ease of use remain key factors in explaining AI adoption in complex Industry 4.0 environments [28]. Similarly, Baroni et al. demonstrate that TAM can be extended for human-in-the-loop AI systems by including AI-specific factors such as trust and perceived quality of AI outputs, while retaining the core TAM structure [29]. In addition, recent research on AI maturity models for engineering explicitly uses technology acceptance as a core dimension to assess and manage AI adoption, further confirming the relevance of TAM even in advanced AI-based engineering applications [30]. These studies show that even when technologies become more advanced, TAM is still applicable for understanding user acceptance.

Like other IT systems and AI-based systems, product configurators also require consideration of end-user perspectives during their development and implementation phases. However, we did not find studies that systematically apply the TAM to product configurators. Based on this gap, we propose a context-specific extension of TAM for product configurators in engineering companies. This model provides an explanatory lens for understanding how users perceive, accept, and use product configurators in practice.

3. Methodology

The main objective of this study is to create a context-specific TAM for product configurators that has both practical relevance for engineering companies and a theoretical contribution to the field of product configurators.

This study follows an interactive research approach. Interactive research is a collaborative form of research in which researchers and practitioners work closely together and create knowledge jointly through continuous interaction, reflection, and iteration between theory and practice [31], [32]. We chose this approach because the phenomenon of interest, namely how users perceive and adopt product configurators, emerges in everyday practice and is difficult to observe from the outside, and close, long-term involvement with a company is needed to capture it. The approach is illustrated in Figure 5.

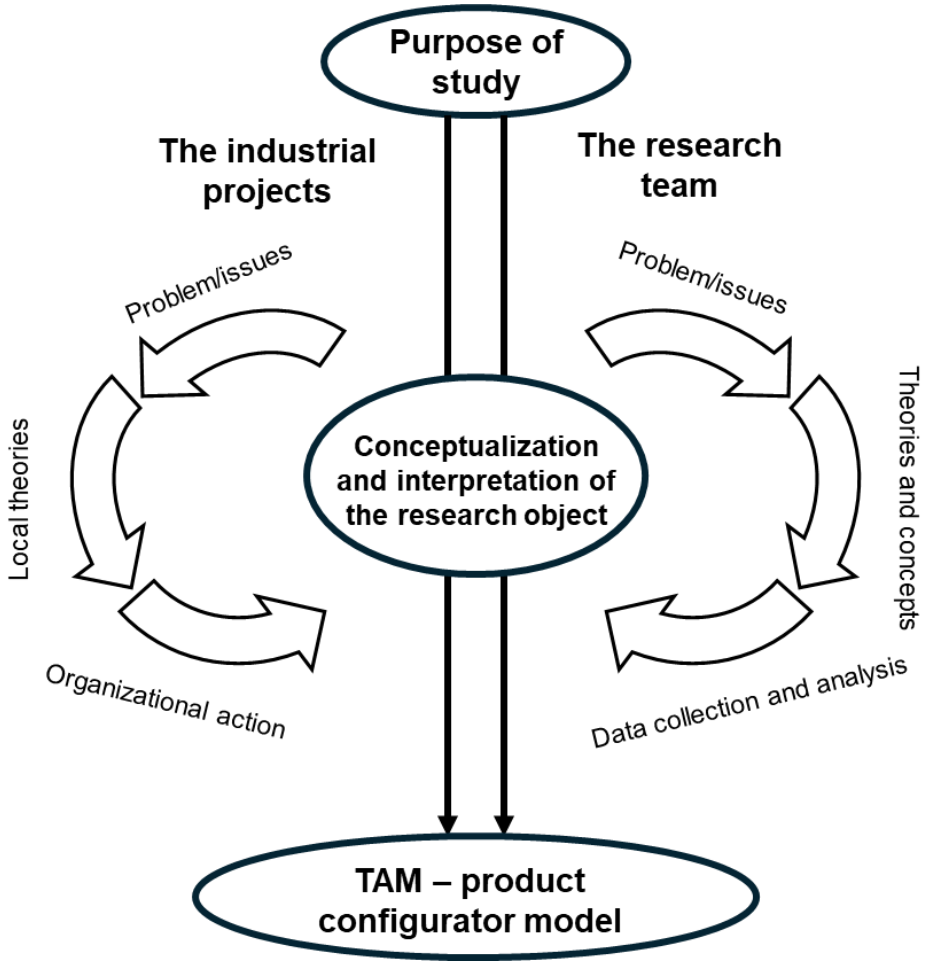


Figure 5: The interactive research approach for this study (redrawn and adapted from [31], [32])

The study is built on a single in-depth case at a leading pharma engineering company in Europe. The case was studied closely from January 2023 to May 2026 through several configurator projects in an engineer-to-order context. These projects involved engineers and managers from different departments and included activities such as workshops, project meetings, informal discussions and pitching, and hands-on involvement in configurator development. Through these activities, practical insights were discussed, collected, and reflected on by both researchers and practitioners. In addition to this main case, the research team draws on its experience from configurator projects in other engineering sectors, including a wind-engineering consultancy, a construction company, a machinery manufacturer, and a plant-engineering company, which served as supporting reference points rather than separate cases.

The author team combined both perspectives. The first author led the model development and was engaged on both the practitioner and the research side; the last author contributed the case-company (practitioner) perspective; the second and third authors, from the research team, contributed expertise in technology acceptance and in configurator projects in engineering,

respectively. The research team also brings years of experience together with literature on product configurators and technology acceptance. These theoretical elements were not applied directly but were used as a reference to interpret and structure the empirical insights.

The knowledge in this study was gained through participatory and largely unstructured engagement rather than through formal, structured interviews. For this reason, the study does not rely on a fixed interview protocol or a formal coding scheme. Instead, recurring observations from the projects were discussed and reflected on by researchers and practitioners and gradually shaped into the factors of the model, drawing on existing TAM constructs where they fit. When existing TAM models or factors were insufficient to explain an observation, new or adapted elements were introduced to better reflect the characteristics of product configurators. Continuous reflection and feedback between researchers and practitioners supported the refinement of the model. Structured interviews, formal qualitative coding, and cross-company validation are deliberately left to future work.

By combining practical experience with established academic knowledge through an interactive research approach, this methodology ensures that the resulting model is both theoretically grounded and closely aligned with real industrial challenges.

4. Results

Based on the interactive research process described above, a TAM-based model for product configurators in engineering companies was developed. Perceived Ease of Use and Perceived Usefulness remain the central constructs in the model. These two factors influence Attitude Towards Using, which then leads to the Actual System Use of the configurator. In addition, the model includes a set of external factors that affect these perceptions, grounded in practical experience from configurator projects in industry and research activities at the university.

The external factors are grouped into three main categories: objective, subjective, and environmental influences. The factors were grouped according to where they originate. Objective factors are properties of the configurator, the product, and the process itself, and are relatively observable and independent of the individual user. Subjective factors relate to the individual user, including their attitudes, experience, and role. Environmental factors lie outside the individual and describe the organizational and market context in which the configurator is used. This separation between technological, individual, and contextual influences is consistent with how external variables are commonly organized in technology-acceptance research.

The objective category covers the configurator IT software, the configurable product, and the configuration process. The subjective category reflects user-related aspects. The environmental category captures broader context conditions, including internal organizational environment and external market environment. These external factors shape how users perceive a configurator as easy to use and useful in their work. These perceptions influence Attitude Towards Using and finally lead to Actual System Use. Figure 6 illustrates the complete model.

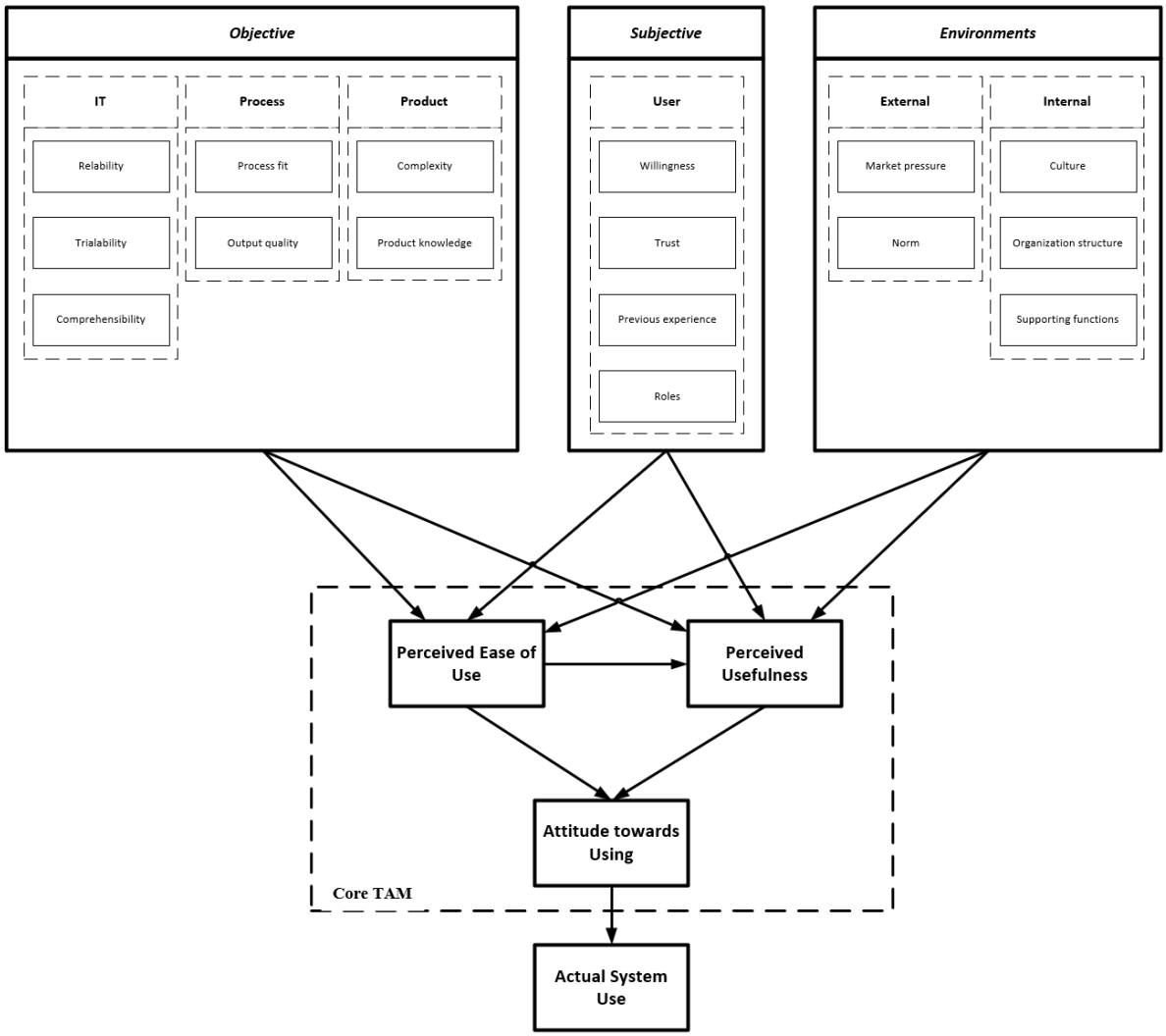


Figure 6: TAM for product configurators in engineering companies

4.1. Objective factors

The first category includes objective factors related to the configurator IT software, the configurable product, and the configuration process it supports.

IT factors play an important role. **Reliability** is critical. The software must be stable and run without system errors or crashes. Otherwise, users lose patience and return to the original working process. **Trialability** also matters. Users are more willing to adopt a configurator if they can test a prototype or limited version before full implementation. **Comprehensibility** is another key aspect. Users are more likely to use and trust the configurator if they understand, at least to some degree, how the software works and how results are generated.

Process factors have a big influence. **Process fit** is essential. A configurator must support the company's actual quotation or design process and allow users to reach the desired results without forcing a complete change of workflow. **Output quality** is another important concern. If the configurator produces frequent errors or results that require constant manual correction, users quickly stop relying on it.

Product factors determine whether users think a configurator is suitable in the first place. **Product structure** affects this strongly. Factors such as product complexity, level of modularity or

standardization, and maturity of the product structure influence whether a product can be configured effectively. Companies often do not have clear views about which products are suitable for configurators and which are not. **Product knowledge** is also important. When knowledge is transparent, maintained, and structured, configurator development and use become much easier. When knowledge is tacit or stored mainly in a few experts' heads, configurator development becomes difficult and users are less confident in the developed configurator.

4.2. Subjective factors

The second category includes subjective factors related to the individual. The individual user's **Willingness** to adopt new tools plays an important role. Some users actively welcome new tools and appreciate the structure and automation that configurators provide. Others prefer traditional methods or rely on their own experience and do not see the configurator as part of their role. **Trust** also influences the use of configurators. Adoption becomes easier when users believe that the configurator brings benefits and has a positive impact. Besides, **Previous experience** also plays a role. Users who have worked with similar IT tools before, either in earlier jobs or within the same company, often accept configurators faster. Good past experiences help adoption, while negative experiences may lead to resistance from the beginning. Finally, **Roles** are another important factor. A configurator may change what users do in their daily work. Some users see the configurator as a way to improve their skills or move into more advanced tasks, while others may feel that their responsibilities are being replaced by automation. How roles are defined before and after configurator implementation, and whether there is a clear development or change plan, strongly affects how users perceive configurators.

4.3. Environmental factors

The final category includes factors inside and outside the user's environment that shape their perception of configurators.

Inside the company, **Culture** plays a strong role. Some companies encourage the use of digital tools, experimentation, and new ways of working. In these cases, configurators are more naturally accepted. In more traditional engineering environments, where individual expertise dominates, experience is highly valued and mistakes are punished, configurators may be viewed as a challenge to established practices. **Organization structure** also influences configurator use. Companies with product-focused structures often have standardized modules and repeatable designs, making configurators easier to introduce. In contrast, project-driven companies typically struggle more, since every project is different and knowledge is harder to structure and maintain. In these situations, configurators require more effort to develop and tend to face more resistance in daily use. **Supporting functions** are another internal factor. The way a company maintains and supports a configurator has a big impact on adoption. Users need access to training, onboarding, updates, and help when problems occur. When support is available and the configurator is followed up through routines and expectations, users are more likely to rely on it. When support is weak and management sees the configurator as optional, the configurator often becomes ignored.

The external environment also shapes adoption. **Market pressure** is often a major driver. When customers or partners expect faster quotations, more accurate design, or more consistent technical data, configurators become a practical necessity rather than a choice. **Norms** in the industry also play a role. If competitors and similar companies actively use configurators, adoption becomes more natural. If configurators are rarely seen in the industry, companies may hesitate or delay investments.

4.4. Core TAM

All external factors play a major role in shaping how users perceive product configurators in terms of ease of use and usefulness. Following Davis [12], we define **Perceived Ease of Use** as how simple and stress-free the configurator feels to work with. This includes how quickly users can learn the configurator, how easy it is to complete their tasks, and how much knowledge is needed to operate it. When the effort required is low, users experience the configurator as easy to use. **Perceived Usefulness** refers to the extent to which users believe that the configurator improves their work performance and brings value to themselves. In practice, this comes down to whether the configurator helps them work faster, make fewer mistakes, produce more accurate results, reduce manual effort, win more orders, make decisions more easily, or move to more value-adding tasks or positions. Perceived ease of use also influences perceived usefulness because a configurator that is simple to operate allows users to experience its benefits more directly, while a difficult configurator hides or reduces those benefits.

Perceived ease of use and perceived usefulness jointly influence the attitude towards using a configurator, which then leads to actual use. Based on experience from both industry projects and our research team, perceived usefulness often has a stronger impact on attitude than ease of use. Users are generally willing to tolerate a system that is difficult or takes effort to learn if they believe it will bring clear benefits. In contrast, even an easy-to-use configurator will not be used if it does not deliver meaningful value. For this reason, usefulness becomes the main driver for adoption, while ease of use acts more as a supporting factor.

Figure 7 shows four scenarios depending on how users perceive usefulness and ease of use. When both are high (**Target**), configurators are more likely to be accepted and integrated into daily work. When usefulness is high, but configurators are not easy to use (**Resistance**), users may understand that the configurator is valuable but struggle with operating it, which leads to complaints or workarounds. When usefulness is low, but it is easy to use (**Ignore**), the developed configurators tend to be ignored, as users do not see enough value in using it. When both are low (**Dead-end**), the configurator is unlikely to be adopted at all.

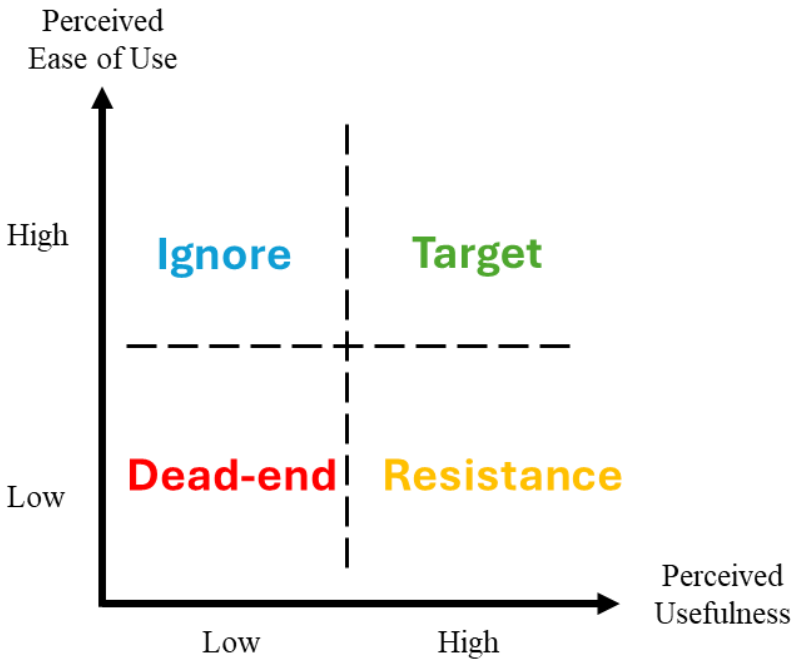


Figure 7: TAM quadrant for product configurators

5. Discussion

Over the centuries, industrial productivity has increased dramatically through the use of new technologies, and all of these technologies have had to be accepted by the people who use them. Challenges of acceptance are not new. A well-known example from the First Industrial Revolution is the Luddite movement, in which English textile workers destroyed mechanized looms in protest against job loss and falling wages [33]. Similar challenges persist today, as IT tools require people to learn new ways of working rather than relying only on past methods.

In the engineering industry, customization has almost become a basic requirement for companies to stay competitive. Together with other IT systems, product configurators make customization easier in engineering companies, which has been well documented in the literature during the past two decades. However, having the technology itself is not enough. Simply developing a configurator does not guarantee that it will be used.

Tools are developed for users, and we must therefore consider the user perspective. By combining technology acceptance theory with insights from industry and research, this study addresses a gap in understanding how users perceive product configurators in engineering companies. This perspective is often overlooked in configurator development projects, which may help explain why some configurators fail to achieve full adoption in practice.

This study has several implications. For research, the model links the configurator literature, which has focused mainly on development frameworks and technical aspects, with technology acceptance theory, and offers a structured way to study why configurators succeed or fail in use. For engineering practice, the factors can be used as a checklist in the early design and planning of configurator projects, where including users as key stakeholders and treating the acceptance factors as key performance indicators can increase the chance that a configurator is adopted in daily work. For management, the model highlights that decisions about culture, organization structure, supporting functions, and roles are as important as the technical scope of a configurator. Although the model is grounded in a single engineering case company, many of the factors are expected to be relevant to other engineering companies, which is an avenue for future work.

This study also has limitations that point to clear next steps. The model is proposed on the basis of a single in-depth case and supporting cross-sector experience. It has not yet been empirically tested, and the external factors are therefore plausible but unranked and unverified. We note that this single-case design is also a source of depth, since many of the factors became visible only through close, long-term involvement with one company. To validate and extend the model, we propose three steps. First, the constructs and external factors should be operationalized into measurable survey items. Second, the resulting hypotheses, including the relative importance and ranking of the external factors and the suggestion that usefulness matters more than ease of use, should be tested through a survey across several engineering companies, complemented by systematically structured interviews. Third, configurator projects should be followed longitudinally, from early development to daily use, to observe how user perceptions change as experience increases. Through this work we hope to encourage broader research attention to the human and organizational factors in configurator adoption.

6. Conclusion

This study shows that technology alone is not enough to ensure successful configurator projects, and that the acceptance of product configurators also depends on the users. To better understand this, we propose a TAM-based model that brings together the factors influencing configurator acceptance in engineering companies. By understanding user perceptions and needs early in

development, companies can increase the success of configurator implementation and avoid building tools that remain unused.

The model is proposed on the basis of a single in-depth case and supporting cross-sector experience, and it has not yet been empirically tested. Future work should therefore validate and extend it through quantitative studies that operationalize and test the factors and their relationships, comparative and cross-sector case studies that examine how the factors transfer to other engineering contexts, and longitudinal studies that follow configurator projects from early development to daily use. We hope this study encourages both practitioners and researchers to integrate user-acceptance thinking into the design, planning, and rollout of product configurators in complex engineering companies.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-5 and Claude Opus 4.8 for grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] Volvo Trucks, "Volvo Trucks leads the heavy truck market in Europe," *Volvo News & insights*, Feb. 19, 2025. Accessed: Jan. 05, 2026. [Online]. Available: https://www.volvotrucks.com/en-en/news-stories/press-releases/2025/feb/volvo-trucks-leads-the-heavy-truck-market-in-europe1.html

[2] Fredrik Elgh, "PhD course - Product Platform," Kunskapsförmedlingen & Jönköping University.

[3] L. Hvam, N. H. Mortensen, and J. Riis, *Product Customization*, 1st ed. Springer Publishing Company, 2008. doi: 10.1007/978-3-540-71449-1.

[4] B. I. Victor and A. C. Boynton, "Making Mass Customization Work," *Harv Bus Rev*, vol. 71, no. 5, pp. 108–119, 1993.

[5] B. J. Pine, Mass customization: the new frontier in business competition. Harvard Business School Press, 1993.

[6] C. W. L. Hart, "Mass customization: conceptual underpinnings, opportunities and limits," *International Journal of Service Industry Management*, vol. 6, no. 2, 1995, doi: 10.1108/09564239510084932.

[7] Volvo Trucks, "Design your Volvo truck," Volvo - Truck Builder. Accessed: Jan. 05, 2026. [Online]. Available: https://www.volvotrucks.com/en-en/trucks/information-and-tools/truck-builder.html

[8] L. Hvam, A. Haug, N. H. Mortensen, and C. Thuesen, "Observed benefits from product configuration systems," *International Journal of Industrial Engineering*, vol. 20, no. 5–6, pp. 329–338, 2013.

[9] A. Haug, L. Hvam, and N. H. Mortensen, "The impact of product configurators on lead times in engineering-oriented companies," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing: AIEDAM*, vol. 25, no. 2, pp. 197–206, May 2011, doi: 10.1017/S0890060410000636.

[10] A. Haug, S. Shafiee, and L. Hvam, "The causes of product configuration project failure," *Comput Ind*, vol. 108, pp. 121–131, Jun. 2019, doi: 10.1016/j.compind.2019.03.002.

[11] M. Møldrup and N. Møller, "Development and implementation of product configuration systems - a change management perspective," in *International Conference on Economic, Technical and Organisational aspects of Product Configuration Systems*, K. Edwards, L. Hvam, M. Møldrup, N.

Møller, J. Riis, and J. L. Pedersen, Eds., Lyngby: Technical University of Denmark, 2004, pp. 47–59. [Online]. Available: http://ipl.dtu.dk

[12] F. D. Davis, “Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information,” *MIS Quarterly*, vol. 13, no. 3, pp. 319–340, 1989.

[13] C. Forza and F. Salvador, “Managing for variety in the order acquisition and fulfilment process: The contribution of product configuration systems,” *Int. J. Production Economics*, vol. 76, pp. 87–98, 2002, doi: 10.1016/S0925-5273(01)00157-8.

[14] A. Haug, “Representation of Industrial Knowledge-as a Basis for Developing and maintaining Product Configurators,” PhD thesis, Technical University of Denmark, 2008.

[15] L. Hvam, “Mass customisation in the electronics industry: based on modular products and product configuration,” *International Journal of Mass Customisation*, vol. 1, no. 4, pp. 410–426, 2006, doi: 10.1504/ijmassc.2006.010442.

[16] C. Wehlin, O. Vidner, L. Poot, and M. Tarkian, “Integrating sales, design and production: A configuration system for automation in mass customization,” in *ASME Design Engineering Technical Conference*, 2021, p. V03BT03A042-V03BT03A042-10. doi: https://doi.org/10.1115/DETC2021-68426.

[17] S. Shafiee, “Conceptual Modelling for Product Configuration Systems,” Technical University of Denmark, Kgs. Lyngby, 2017.

[18] Q. Zhao and L. Hvam, “Two Sides of One Coin: Aligning Configuration System with Product Architecture for ETO Products,” in *27th INTERNATIONAL DEPENDENCY AND STRUCTURE MODELING CONFERENCE*, 2025, pp. 24–26. doi: 10.35199/dsm2025.08.

[19] Q. Zhao and L. Hvam, “Framework for Investigating Possible Areas to Apply Product Configuration Systems For Engineer-To-Order Products,” in *11th International Conference on Customization and Personalization MCP 2024*, University of Novi Sad, 2024, pp. 265–272.

[20] S. Shafiee, L. Hvam, and M. Bonev, “Scoping a Product Configuration Project for Engineer-to-Order Companies,” *International Journal of Industrial Engineering and Management (IJIEM)*, vol. 5, no. 4, pp. 207–220, 2014.

[21] K. Kristjansdottir, S. Shafiee, and L. Hvam, “How to Identify Possible Applications of Product Configuration Systems in Engineer-to-Order Companies,” *International Journal of Industrial Engineering and Management (IJIEM)*, vol. 8, no. 3, pp. 157–165, 2017.

[22] A. Haug, “The illusion of tacit knowledge as the great problem in the development of product configurators,” *Artificial Intelligence for Engineering Design, Analysis and Manufacturing: AIEDAM*, vol. 26, no. 1, pp. 25–37, Feb. 2012, doi: 10.1017/S0890060410000533.

[23] K. R. Ladeby and G. V Oddsson, “Reference frame for Product Configuration,” in *2011 World Conference on Mass Customization, Personalization, and Co-Creation*, 2011.

[24] V. Venkatesh, M. G. Morris, G. B. Davis, and F. D. Davis, “User Acceptance of Information Technology: Toward a Unified View,” *MIS Quarterly*, vol. 27, no. 3, pp. 425–478, 2003.

[25] V. Venkatesh and F. D. Davis, “Theoretical extension of the Technology Acceptance Model: Four longitudinal field studies,” *Manage Sci*, vol. 46, no. 2, pp. 186–204, 2000, doi: 10.1287/mnsc.46.2.186.11926.

[26] P. Legris, J. Ingham, and P. Collerette, “Why do people use information technology? A critical review of the technology acceptance model,” *Information & management*, vol. 40, no. 3, pp. 191–204, 2003.

[27] N. Marangunić and A. Granić, “Technology acceptance model: a literature review from 1986 to 2013,” *Univers Access Inf Soc*, vol. 14, no. 1, pp. 81–95, Mar. 2015, doi: 10.1007/s10209-014-0348-1.

[28] S. Chatterjee, N. P. Rana, Y. K. Dwivedi, and A. M. Baabdullah, “Understanding AI adoption in manufacturing and production firms using an integrated TAM-TOE model,” *Technol Forecast Soc Change*, vol. 170, Sep. 2021, doi: 10.1016/j.techfore.2021.120880.

[29] I. Baroni, G. Re Calegari, D. Scandolari, and I. Celino, “AI-TAM: a model to investigate user acceptance and collaborative intention in human-in-the-loop AI applications,” *Human Computation*, vol. 9, no. 1, pp. 1–21, May 2022, doi: 10.15346/hc.v9i1.134.

[30] O. Bleisinger, C. Gewalt, J. Heinrich, and M. Eigner, “Assessment of AI-Adoption in Product Development by a Maturity Model Based on Technology Acceptance Scoring,” in *2025 IEEE International Conference on Industrial Engineering and Engineering Management*, IEEE, 2025.

[31] L. Ellström and P.-E. Brulin, “Introduction-on interactive research,” 2007. [Online]. Available: www.Hampp-Verlag.de

[32] A. Engström, A. Johansson, N. Edh Mirzaei, K. Sollander, and D. Barry, “Knowledge creation in projects: an interactive research approach for deeper business insight,” *International Journal of Managing Projects in Business*, vol. 16, no. 1, pp. 22–44, Mar. 2023, doi: 10.1108/IJMPB-09-2021-0233.

[33] E. J. Hobsbawm, “The Machine Breakers,” *Past Present*, no. 1, pp. 57–70, 1952, [Online]. Available: https://www.jstor.org/stable/649989?seq=1&cid=pdf-

Enabling Automated Usability Evaluation for Configurator User Interfaces

Sebastian Lubos[1,*], Alexander Felfernig[1], Viet-Man Le[1], Thi Ngoc Trang Tran[1], Doris Suppan[2], Reinhard Willfort[2], Ivan Dukic[3], Jeremias Fuchs[3] and Manuel Henrich[4]

[1]*Graz University of Technology, Inffeldgasse 16b, Graz, 8010, Austria*
[2]*isn – innovation service network GmbH, Grabenstraße 231, Graz, 8045, Austria*
[3]*Morgendigital GmbH, Jahnstraße 18, Innsbruck, 6020, Austria*
[4]*UNiQUARE Software Development GmbH, Lannerweg 9, Krumpendorf am Wörthersee, 9201, Austria*

Abstract

Configurators are used by customers to personalize products and services. These rely on robust domain models for performance and technical correctness. However, the *usability* of configurator *user interfaces (UIs)* is often overlooked as a quality attribute, even though it directly affects the ability of customers to complete configurations. Manual heuristic evaluations conducted by experts and other methods can uncover usability issues, but are infrequently used in many companies due to high cost, limited time, and a lack of expertise, as revealed in a survey among practitioners. Also, traditional usability heuristics focus on general usability aspects and fail to address configurator-specific interactions. In this paper, we present a novel tool that automatically identifies usability issues and recommends explicit improvements for configurator UIs using *multimodal large language models (MLLMs)* and screen recordings of user interactions as context. The tool uses configurator-specific criteria to generate reviews. This helps to reduce the effort of manual analysis and makes usability evaluation accessible to users with limited usability expertise.

Keywords

Configuration, Usability Evaluation, Multimodal Large Language Models, Configurator User Interface

1. Introduction

Configurators enable customers to personalize products and services based on their individual preferences and needs [1, 2]. They usually serve as the main entry point for placing orders [3]. To ensure that customers can complete configurations, domain knowledge must be correctly encoded, and configurator performance must be adequate [2, 4, 5]. However, one quality aspect that frequently receives less attention is the *usability* of configurator *user interfaces (UIs)*, which help users understand available options, express preferences, explore alternatives, construct valid solutions, and complete orders [6]. The *ISO9241-11:2018* standard defines *usability* as the extent to which users achieve specified goals effectively, efficiently, and satisfactorily [7]. This means that even when configurators produce technically valid configurations, poor usability can lead to unfulfilled customer expectations [8]. Therefore, good usability is highly relevant for the successful completion of configuration sessions, especially since end users often have limited domain expertise.

While practitioners in software development companies acknowledge the importance of usability in product development, they recognize multiple reasons why usability evaluations are not conducted regularly (see Section 2). These include a lack of time and expertise, as well as high cost.

Traditional methods to evaluate the usability of systems include, for example, *usability tests*, in which real users are observed while solving tasks [9], or *usability inspections* [10], in which usability experts

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany
*Corresponding author.
sebastian.lubos@tugraz.at (S. Lubos); alexander.felfernig@tugraz.at (A. Felfernig); v.m.le@tugraz.at (V. Le); trang.tran@tugraz.at (T. N. T. Tran)
0000-0002-5024-3786 (S. Lubos); 0000-0003-0108-3146 (A. Felfernig); 0000-0001-5778-975X (V. Le); 0000-0002-3550-8352 (T. N. T. Tran); 0009-0008-6410-0947 (R. Willfort); 0009-0002-5907-5173 (J. Fuchs); 0009-0007-4644-0836 (M. Henrich)
© 2026 This work is licensed under a "CC BY 4.0" license.

simulate how real users solve tasks. While both methods aim to identify usability issues, they are resource-intensive due to manual analysis and the need for expert knowledge.

To address this, various approaches to automate usability evaluations have been proposed, including heuristic checkers and rule-based methods [11, 12]. However, these typically evaluate only limited aspects and contexts [13]. Recent methods rely on *multimodal large language models (MLLMs)* and report promising results [14, 15, 16]. MLLMs process textual descriptions alongside visual representations [17]. For usability evaluation, the general approach is to include textual evaluation instructions and to use screenshots or screen recordings of user interactions of the evaluated application as visual context.

This paper builds on prior research and presents the first tool that enables automated usability evaluation of configurator applications. Based on uploaded screen recordings of user interactions, our tool runs an MLLM-based usability evaluation to identify issues, with severity ratings and explicit recommendations to improve them. For this purpose, the tool applies configurator-specific usability criteria derived from existing literature (see Section 3). These criteria consider configurator-specific challenges that general heuristics do not cover, such as complex compatibility constraints or large option spaces, and deliver additional benefit for the usability evaluation of configurator UIs. The tool presents the evaluation results in a structured way to enable reviewing and validating identified issues and suggested improvements. Overall, this paper makes the following contributions:

- We present the results of a survey among software development practitioners to understand the importance and challenges of usability evaluation in their development processes.
- We introduce the first tool for MLLM-based usability evaluation that applies criteria specifically targeted at user interactions with configurator UIs.

2. Practitioner Survey

To understand the relevance of usability evaluations for companies and the challenges they face when applying these methods, we conducted a survey among practitioners to learn about their requirements in this area. Participants could complete the survey in German or English, and invitations were sent via mail and LinkedIn. The participation was voluntary and not compensated. Overall, 24 participants answered the survey and shared their feedback.

Participants represented companies of all sizes, evenly distributed across five employee-count categories (< 10, $10 - 49$, $50 - 249$, $250 - 999$, ≥ 1000). As shown in Tables 1a and 1b, their developed products were primarily web, desktop, and mobile applications, while a broader spectrum of end users was represented. Overall, these properties reflect important contexts for usability evaluations.

Table 1
Details about the company products of survey participants (multiple choice).

Product Type	Responses
Web applications	18 (75%)
Desktop software	9 (38%)
Mobile applications	8 (33%)
Embedded/hardware interfaces	2 (8%)
Other	2 (8%)
Physical products	1 (4%)

(a) Developed product types.

Product End Users	Responses
Business users/Professionals	14 (58%)
End users/Consumers	12 (50%)
Internal staff	10 (42%)
Administrators/IT	6 (25%)
Partners/Resellers	2 (8%)

(b) End users of developed products.

Furthermore, participants were asked about the perceived importance of usability evaluations and the frequency with which they are conducted in their company. The results point to a gap between attitude and practice (see Table 2). Although 54% of respondents rate usability evaluation as essential or important, 71% conduct evaluations only once a year or less frequently. More than half of those who believe it is important do not evaluate more than once a year. This pattern also aligns with the

organizational responsibility for usability in practice, as nearly half of the companies have no assigned responsibility for this topic (see Table 3a).

Table 2
Perceived importance of usability evaluation by evaluation frequency in companies.

	Essential/Important	Neutral	Unimportant	Total Frequency
At least monthly	6 (25%)	0 (0%)	1 (4%)	7 (29%)
Yearly or less	7 (29%)	4 (17%)	6 (25%)	17 (71%)
Total Importance	13 (54%)	4 (17%)	7 (29%)	24 (100%)

To understand the reasons for these discrepancies, participants were asked to specify all applicable barriers they perceive (see Table 3b). The most prominent reasons were a lack of time, internal priorities, budget constraints, and insufficient skills/know-how. These responses indicate that operational constraints limit the inclusion of usability evaluation in practice, while its value is recognized. This motivates our focus on automation to make usability evaluations more accessible and routine within existing development workflows.

Table 3
Organizational responsibility and reasons for infrequent usability evaluations in companies (multiple choice).

Assigned responsibility	Responses
No specific competence	11 (46%)
Own UX team	7 (29%)
Individual UX roles	4 (17%)
External contractors	2 (8%)

(a) Organizational responsibility of usability and UX.

Reasons	Responses
Lack of time	12 (50%)
Low internal priority	11 (46%)
Lack of budget	10 (42%)
Lack of skills/know-how	7 (29%)
No access to end-users	5 (21%)
Other methods are sufficient	3 (13%)
Unclear benefit/ROI	2 (8%)

(b) Reasons for infrequent usability evaluations.

3. Configurator-Specific Usability Criteria

Different usability guidelines support the evaluation of interactive software applications. These include general heuristics, such as the *ISO9241-110* interaction principles [18] and *Nielsen heuristics* [19], which address general concepts of human-computer interaction and can therefore be applied to most applications. Other guidelines are platform-dependent and more design-oriented, for example, Android's *UI Design* best practices[1] and Apple's *Human Interface Guidelines*[2]. Furthermore, the *Web Content Accessibility Guidelines (WCAG)*[3] consider accessibility-related aspects to ensure usage by all users, including those with disabilities. While all of these criteria are also relevant to configurator UIs, they do not account for configurator-specific user interactions.

Lubos et al. [16] suggested a set of configurator-specific usability criteria derived from related research. The criteria are summarized in Tables 4–7, each with a detailed description that can be used to evaluate the extent to which a configurator application fulfills each criterion. The criteria are grouped by the specific parts of the user-configurator interaction they address:

- **Configuration Process** summarizes the criteria for the preference specification, including the availability and presentation of configuration options, the comparison of alternatives, and the prevention of invalid configurations (see Table 4).
- **Navigation** considers how users can navigate efficiently, transparently, and flexibly, based on their individual needs (see Table 5).

[1] https://developer.android.com/design/ui
[2] https://developer.apple.com/design/human-interface-guidelines
[3] https://www.w3.org/TR/WCAG21/

- **Explanation** assesses the help for users to understand the product domain, dependencies, and gives support if errors occur (see Table 6).
- **Visualization** reviews the possibilities of users to review whether the configuration adequately fulfills their needs (see Table 7).

Table 4
Evaluation criteria of the *configuration process* category.

ID	Criterion	Description	References
C1	Customized options	Does the configurator adapt the available options to meet different user profiles (e.g., expert vs. non-expert), such as presenting a needs-based view or a parameter view, with a clear way to select the profile?	[6, 20, 21]
C2	Organized configuration space	Does the configurator help users with large option spaces by utilizing explicit mechanisms (e.g., grouping, search/filtering, multi-step configuration) to keep the number of simultaneously visible choices manageable?	[8, 20]
C3	Availability of options	Can users revisit and change previously set options without losing their current configuration state?	[3, 8, 20]
C4	Auto-completion	Does the configurator offer user-triggered auto-completion that fills in the remaining required options with defaults?	[3]
C5	Variant comparison	Can users keep multiple variants and compare them (e.g., ranking by properties, side-by-side, or highlighting differences) to review trade-offs?	[6, 21]
C6	Error prevention	Does the configurator prevent users from ending up with an invalid configuration (e.g., by disabling incompatible options, auto-resolving conflicts, or requiring conflict resolution before proceeding)?	[3, 8, 22]

Table 5
Evaluation criteria of the *navigation* category.

ID	Criterion	Description	References
N1	Focused navigation	Does the configurator provide a clear, task-aligned sequence of steps that helps users reach a valid result efficiently (e.g., using a relevant order of decisions or avoiding unnecessary steps)?	[3, 8, 22, 23]
N2	Manual step transition	Does the configurator support moving forward/backward using consistently placed, clearly labeled controls?	[3, 8]
N3	Flexible navigation	Can users modify or undo earlier selections without losing the current configuration state?	[3, 8, 21]
N4	Progress indication	Does the configurator clearly indicate the current step, remaining steps, and completion status (e.g., stepper or progress bar)?	[3, 22]
N5	Variant persistence	Can users save, name, or restore previous full configuration variants during or after the session (e.g., version history or saved configurations)?	[21]
N6	Starting points	Does the system offer practical starting points to support different user needs (e.g., starting from predefined configurations or selecting properties to start with)?	[6, 21]

These configurator-specific usability criteria encompass a broad range of aspects that can help to improve the quality of user interaction with configurators. As some of the criteria are tightly coupled to specific features of a configurator, e.g., the *auto-completion* of configurations (see C4 in Table 4), it is important to notice that their evaluation is beyond simply identifying whether these features are present. The focus is also on how intuitive they are to use. In this sense, the availability of a feature does not necessarily indicate that it is intuitive to use and sufficiently supports users in their interactions.

4. Automated Usability Evaluation

Automating usability evaluations makes them more accessible to companies. Especially for complex applications such as configurators, good usability is essential for customers to complete configurations. In the following, we describe our tool that uses an MLLM to detect usability issues from screen recordings of user interactions, following [15]. The overall tool flow is depicted in Figure 1 and contains *five* steps:

Table 6
Evaluation criteria of the *explanation* category.

ID	Criterion	Description	References
E1	Providing domain knowledge	Does the configurator provide in-context explanations of options (e.g., tooltips or examples) that clarify meaning and help users make informed choices?	[3, 6, 21, 23]
E2	Transparency of dependencies	Does the configurator explain dependencies of options at decision time (e.g., 'choosing X requires Y', 'this removes Z', impact on price/delivery) to highlight consequences of choices?	[6, 21, 22]
E3	Transparency of errors	Does the configurator explain why a configuration is inconsistent (e.g., highlighting which selected options are in conflict) using actionable, non-technical language?	[3, 6, 8, 20, 22]
E4	Repair suggestions	Does the configurator suggest actionable repair options (e.g., 'change X to Y or Z' or 'remove X') to assist users in resolving errors with inconsistent constraints?	[3, 6, 8]

Table 7
Evaluation criteria of the *visualization* category.

ID	Criterion	Description	References
V1	Product preview	Does the configurator provide a product preview that is continuously updated after changes to reflect the current configuration?	[3, 6, 8, 22, 23]
V2	Customized preview	Can users switch between preview modes (e.g., 2D/3D images or a textual summary), with a consistent relation to the current configuration?	[20, 23]

1. **Data Preparation**: Create a screen recording of user interactions with the configurator. Using screen recordings instead of screenshots helps preserve interaction dynamics and UI state transitions, enabling a more comprehensive usability analysis.
2. **Upload**: Upload the recording. An optional description that describes its usage context is provided for more narrow issue descriptions. For example, stating that end users are IT administrators prevents flagging domain-specific UI labels that might only be unclear to a broad audience.
3. **MLLM-based Usability Evaluation**: The MLLM evaluates the application based on the uploaded context and specified usability criteria (see Tables 4–7). The general role and boundaries are specified in the system prompt (see Listing 1), while the user prompt includes the criterion-specific instructions (see Listing 2). For each criterion,[4] the MLLM returns a structured JSON response that includes a list of identified issues per criterion, where each issue consists of an *issue description*, *severity rating*, and *improvement recommendation*.
4. **Result Review**: Review the identified usability issues, grouped by the violated usability criterion and issue severity. The severity helps prioritize issues that should be resolved quickly while avoiding forgetting those with lower impact.
5. **Validation and Feedback**: Choose to accept or reject each issue to enable human validation. If an issue is rejected, a reason can be selected from a dropdown. This feedback is used to improve the tool's accuracy in future updates.

```
You are a Senior Usability Engineer specializing in evidence-based UI/UX audits.
Your task is to perform an evaluation of software interfaces based strictly on provided visual evidence and context.

Core Principles:
1. Strict Visual Evidence: Report ONLY what is explicitly visible in the screenshots or recording.
2. No "Missing" Hypotheses: You are strictly forbidden from reporting the absence of features (e.g., "missing
     confirmation dialogs" or "lack of undo options") unless the recording explicitly shows a user attempting a specific
      action where that feature is expected and fails to appear.
3. No Assumptions: Do not speculate on what happens before or after the provided media. If the recording ends, do not
     assume the next step is missing or broken.
4. Data Neutrality: Ignore the actual values of the data shown (e.g., placeholder text, test names like "Test User", or
     mock prices like "$0.00"). Assume the data is intentional for the test environment.
```

[4]We consider one criterion per prompt for more focused and precise results instead of evaluating multiple criteria at once.

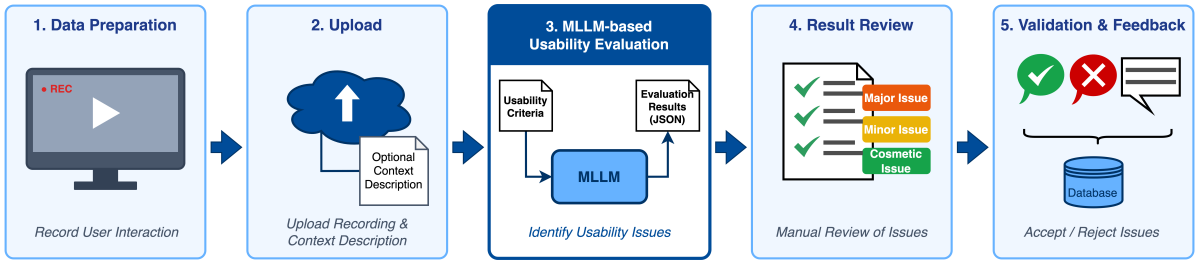


Figure 1: Overview of the automated usability analysis using an MLLM.

```
5. Interface Focus: Focus exclusively on UI/UX elements: layout, navigation, visual hierarchy, affordance, and
     interaction logic. Report on how data is presented (e.g., alignment, legibility, wrapping), not the substance of
     the data itself.
6. Actionable Precision: Recommendations must be specific enough for a developer to implement  (e.g., specify placement,
     contrast, or labeling).
7. Tone: Professional, objective, and analytical.
```

Listing 1: System prompt for the usability evaluation, providing general instructions and boundaries.

```
Usability Criterion Under Evaluation: {criterion_name}
Usability Criterion Explanation: {criterion_description}
{application_context}

Evaluation Task
Analyze the provided interface against the criterion above.
Base your findings solely on visible friction points.

Follow these steps:
1. Identify any usability issues related to the criterion and explain each issue.
2. Provide a practical, actionable recommendation to fix each issue.
3. Assign a severity level.

Severity Scale:
- 1: Cosmetic - Minor aesthetic issues or very small frustrations.
- 2: Minor - Usability problems that slow down the user but don't prevent task completion.
- 3: Major - Important usability problems that lead to significant frustration or potential task abandonment.
- 4: Catastrophe - Critical issues that prevent the user from completing their task or cause data loss.
[...]
```

Listing 2: Prompt template for the usability evaluation with variable placeholders indicated in braces "{}". We use the severity scale suggested by *Nielsen*.[5] Result formatting instructions are omitted.

The current version uses Google's *gemini-2.5-flash* MLLM for the usability evaluation [24], which efficiently handles multimodal inputs, including video. To make outputs more deterministic, the *temperature* parameter was set to 0. The screen recordings are directly submitted to the API using the default settings, which extract one frame per second.[6] This balances *cost* and *detail*, as fewer frames lower token usage[7] and speeds up analysis, but may miss relevant interactions. In contrast, more frames add detail at a higher cost. Reaching the context-token limit is unlikely with typical user interaction samples, as at least 1 hour of video can be processed with default settings before exceeding it.

5. Practical Example and Discussion

We demonstrate the tool by evaluating the usability of the *AimControllers* configurator.[8] *AimControllers allows gamers to fully customize and purchase modified gaming controllers by configuring aesthetic elements alongside functional upgrades.*[9] We created a screen recording simulating a typical configuration session and uploaded it to the tool. After a few minutes, it presented the results as shown in Figure 2.

[5] https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/
[6] https://ai.google.dev/gemini-api/docs/video-understanding
[7] Each extracted frame contributes 258 tokens under default settings.
[8] https://eu.aimcontrollers.com
[9] This sentence was also used as the app description in the tool.

The tool identified *18* usability issues across the configurator-specific criteria with different severities. The UI shows the uploaded video on the left and an issue list on the right, grouped by severity and violated criterion. The issues can be filtered by severity, and improvement suggestions are collapsible to reduce text. The criterion definitions are available via tooltips. Users can accept or decline issues using the thumbs-up or thumbs-down icons. In the latter case, they are hidden in the list after the user provides the reason. In addition to sharing the result page, a PDF summary can be downloaded.

This work focuses on the presentation of an accessible tool that enables automated usability evaluations of configurators. An in-depth evaluation of the quality of identified issues and improvement suggestions using general usability heuristics was conducted in [15], while a configurator-specific study across 16 commercial configurators from different domains confirmed that most identified issues were plausible and accompanied by helpful improvement suggestions [16]. The latter study also highlighted distinct trends regarding which configurator usability criteria (see Section 3) are widely adopted and where issues were frequently recognized. The most frequently fulfilled criteria included *availability of options (C3)*, *transparency of errors (E3)*, *manual step transitions (N2)*, *starting points (N6)*, and *product preview (V1)*. In contrast, *customized options (C1)*, *variant comparison (C5)*, *providing domain knowledge (E1)*, *progress indication (N4)*, and *variant persistence (N5)* appeared to be commonly overlooked. Automated usability evaluation can help reveal these issues in configurators and support their resolution in commercial configurators, thereby improving the user experience.

For the *AimControllers* example, one author of this paper reviewed the identified issues for plausibility in terms of describing actual issues in the application and considered most of them (12/18) plausible. The implausible issues in this example were mainly due to domain misunderstandings, such as misclassifying valid option combinations as incompatible or misinterpreting the positioning of labels in the UI. Reducing these false positives likely requires prompt tuning and improvements in the provision of visual context. As no benchmark datasets for usability evaluations exist, a curated dataset with example applications and validated issues is needed to automatically test the accuracy of approaches and models. For now, human validation remains essential as suggested by related studies [14, 15, 16]. To address this, our tool includes built-in controls that allow users to accept or decline issues and provide feedback, helping to collect validation data for continuous system improvement.

Beyond textual recommendations, automated resolution could suggest concrete UI code changes if the configurator's code repository is accessible [25]. This way, the MLLM acts as an assistant, translating usability findings into implementation-level guidance for developers.

Overall, MLLM-based usability evaluation offers practical support that can make its adoption in the industry more feasible. By enabling accessible, semi-automated evaluations, also for complex configurator applications, this approach directly addresses barriers identified in our practitioner survey,

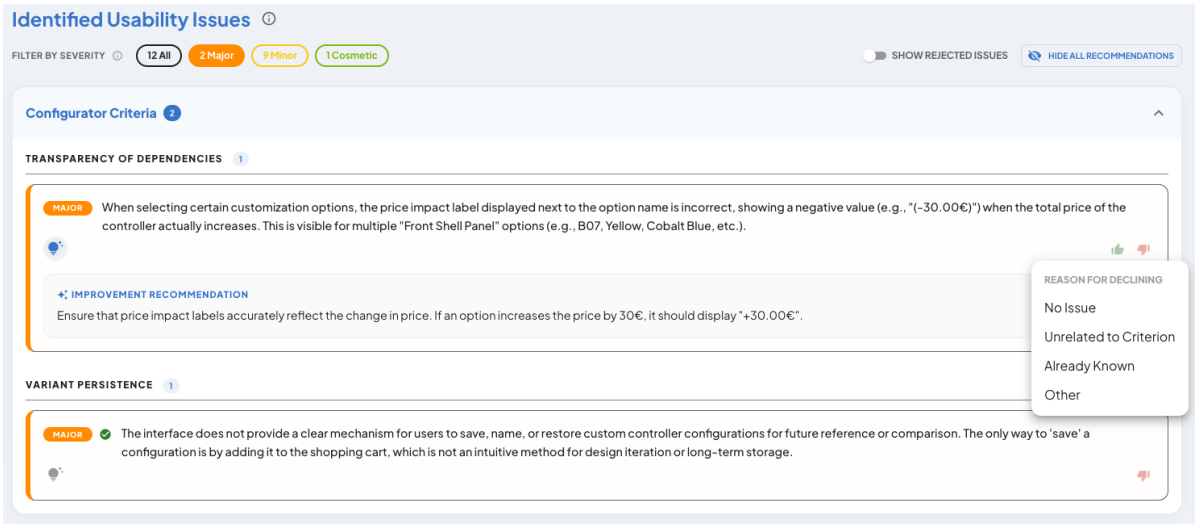


Figure 2: Tool screenshot showing identified usability issues for the *AimControllers* application. In the full UI, the uploaded screen recording appears to the left of these issues.

such as a lack of time, budget, and know-how (see Section 2).

6. Conclusions

This work presented a novel tool that automates the usability evaluation of configurator UIs by analyzing screen recordings of user interactions with *multimodal large language models (MLLMs)*. Using configurator-specific criteria, it generates structured issue descriptions with severity ratings and explicit improvement recommendations, reducing the effort and expertise required for typical usability evaluations. The approach is motivated by a survey we conducted among practitioners. It highlights that although the importance of usability is widely recognized, evaluations are infrequent due to constraints on time, budget, internal priorities, and know-how. This motivates automated analysis with human-in-the-loop validation, as supported by our tool. Future work aims to improve its accuracy by refining prompts, enabling agent-driven exploration to further reduce manual effort, and extending the improvement recommendations to provide direct implementation-level guidance.

Acknowledgments

The presented work has been developed within the research project GENRE, which is funded by the Austrian Research Promotion Agency (FFG) under the project number 915086.

Declaration on Generative AI

The authors used *GPT-5* and *Grammarly* for *grammar and spelling checks* and to *paraphrase and reword*. The authors reviewed and edited the content and take full responsibility for its publication.

References

[1] A. Felfernig, Standardized configuration knowledge representations as technological foundation for mass customization, IEEE Transactions on Engineering Management 54 (2007) 41–56. doi:10.1109/TEM.2006.889066.

[2] A. Felfernig, A. Falkner, D. Benavides, Analysis of Feature Models, Springer International Publishing, Cham, 2024, pp. 45–72. doi:10.1007/978-3-031-61874-1_3.

[3] E. K. Abbasi, A. Hubaux, M. Acher, Q. Boucher, P. Heymans, The anatomy of a sales configurator: an empirical study of 111 cases, in: Proceedings of the 25th International Conference on Advanced Information Systems Engineering, CAiSE'13, Springer-Verlag, Berlin, Heidelberg, 2013, p. 162–177. doi:10.1007/978-3-642-38709-8_11.

[4] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, G. Saake, Predicting performance via automated feature-interaction detection, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, 2012, p. 167–177.

[5] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A classification and survey of analysis strategies for software product lines, ACM Comput. Surv. 47 (2014). doi:10.1145/2580950.

[6] G. Leitner, A. Felfernig, P. Blazek, F. Reinfrank, G. Ninaus, Chapter 8 - user interfaces for configuration environments, in: A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen (Eds.), Knowledge-Based Configuration, Morgan Kaufmann, Boston, 2014, pp. 89–106. doi:10.1016/B978-0-12-415817-7.00008-6.

[7] International Organization for Standardization, ISO/IEC/IEEE International Standard - Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts, ISO/IEC/IEEE 9241-11:2018(E) (2018).

[8] R. Rabiser, P. Grünbacher, M. Lehofer, A qualitative study on user guidance capabilities in product configuration tools, in: Proceedings of the 27th IEEE/ACM International Conference on Automated

Software Engineering, ASE '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 110–119. doi:10.1145/2351676.2351693.

[9] C. Hass, A Practical Guide to Usability Testing, Springer International Publishing, Cham, 2019, pp. 107–124. doi:10.1007/978-3-319-96906-0_6.

[10] T. Hollingsed, D. G. Novick, Usability inspection methods after 15 years of research and practice, in: Proceedings of the 25th Annual ACM International Conference on Design of Communication, SIGDOC '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 249–255. doi:10.1145/1297144.1297200.

[11] J. W. Castro, I. Garnica, L. A. Rojas, Automated tools for usability evaluation: A systematic mapping study, in: G. Meiselwitz (Ed.), Social Computing and Social Media: Design, User Experience and Impact, Springer International Publishing, Cham, 2022, pp. 28–46.

[12] A. Namoun, A. Alrehaili, A. Tufail, A review of automated website usability evaluation tools: Research issues and challenges, in: M. M. Soares, E. Rosenzweig, A. Marcus (Eds.), Design, User Experience, and Usability: UX Research and Design, Springer International Publishing, Cham, 2021, pp. 292–311.

[13] E. Kuric, P. Demcak, M. Krajcovic, J. Lang, Systematic literature review of automation and artificial intelligence in usability issue detection, 2025. doi:10.48550/arXiv.2504.01415.

[14] A. E. Pourasad, W. Maalej, Does GenAI Make Usability Testing Obsolete? , in: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE Computer Society, Los Alamitos, CA, USA, 2025, pp. 675–675. doi:10.1109/ICSE55347.2025.00138.

[15] S. Lubos, A. Felfernig, D. Garber, V.-M. Le, M. Henrich, Recommending usability improvements with multimodal large language models, Proc. ACM Softw. Eng. 3 (2026). doi:10.1145/3797121.

[16] S. Lubos, A. Felfernig, D. Garber, A. Kraljić, T. Kraljić, V.-M. Le, T. N. T. Tran, G. Leitner, J. Schwazer, D. Suppan, R. Willfort, I. Dukic, J. Fuchs, M. Henrich, Usability analysis of configurator user interfaces with multimodal large language models, 2026. doi:10.48550/arXiv.2605.29456.

[17] S. Yin, C. Fu, S. Zhao, K. Li, X. Sun, T. Xu, E. Chen, A survey on multimodal large language models, National Science Review 11 (2024). doi:10.1093/nsr/nwae403.

[18] International Organization for Standardization, ISO/IEC/IEEE International Standard - Ergonomics of human-system interaction – Part 110: Usability: Interaction Principles, ISO/IEC/IEEE 9241-110:2020(E) (2020).

[19] J. Nielsen, Enhancing the explanatory power of usability heuristics, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '94, Association for Computing Machinery, New York, NY, USA, 1994, p. 152–158. doi:10.1145/191666.191729.

[20] T. Leclercq, E. K. Abbasi, B. Dumas, M.-A. Remiche, P. Heymans, Essential expectations of users of web configurators: An empirical survey, Proc. ACM Hum.-Comput. Interact. 6 (2022). doi:10.1145/3534519.

[21] A. Trentin, E. Perin, C. Forza, Sales configurator capabilities to avoid the product variety paradox: Construct development and validation, Computers in Industry 64 (2013) 436–447. doi:10.1016/j.compind.2013.02.006.

[22] T. Leclercq, M. Cordy, B. Dumas, P. Heymans, On studying bad practices in configuration uis, in: Joint Proceedings of the ACM IUI 2018 Workshops, 2018. URL: https://ceur-ws.org/Vol-2068/wii1.pdf.

[23] T. Rogoll, F. Piller, Product configuration from the customer's perspective: A comparison of configuration systems in the apparel industry, in: Proceedings of the International Conference on Economic, Technical and Organisational aspects of Product Configuration Systems (PETO 2004), Lyngby, Denmark, 2004, pp. 179–199.

[24] Gemini Team Google, Gemini: A family of highly capable multimodal models, 2024. doi:10.48550/arXiv.2312.11805.

[25] J. Jiang, F. Wang, J. Shen, S. Kim, S. Kim, A survey on large language models for code generation, ACM Trans. Softw. Eng. Methodol. 35 (2026). doi:10.1145/3747588.

Cognitive Robustness of Large Language Models in Configuration Knowledge Engineering

Viet-Man Le, Sebastian Lubos, Thi Ngoc Trang Tran and Alexander Felfernig

Institute of Software Engineering and AI, Graz University of Technology, Graz, Austria

Abstract

Cognitive aspects of constraint representations have been studied in configuration knowledge engineering. Earlier work documented that human knowledge engineers make systematic errors in three situations: when working with logically equivalent constraint forms, when constraints are presented in different orders, and when translating between natural language statements and formal encodings. We ask whether these human error patterns transfer when large language models (LLMs) are the test subjects, how they extend to real-world configuration knowledge bases, and whether the domain meaning of feature names helps or hurts LLM reasoning. Five LLMs are tested across nine experimental settings: four replicate the human findings on small synthetic constraint satisfaction problems, four extend the same manipulations to feature model (FM) knowledge bases of 13 to 169 features, and one examines whether the knowledge encoded in pretrained LLMs influences their reasoning when knowledge bases contain domain-meaningful feature names. Across these experiments, we find that human error patterns largely do not transfer to LLMs: the representation, ordering, and formalization manipulations that derail human knowledge engineers leave all five LLMs highly accurate, and on redundancy detection three of the five outperform the human baseline. However, three LLM-specific vulnerabilities emerge: (1) when asked to produce a valid configuration, GPT-4.1 mini collapses as FM size grows, while the four other LLMs are unaffected; (2) domain-meaningful feature names lower the tested LLMs' accuracy compared to abstract identifiers (`f1`...`fN`); and (3) the *mandatory* relationship is not reliably identified in natural-language descriptions of FMs, where the LLMs conflate it with cross-tree *requires* and *or* relationship. Taken together, the cognitive-robustness profile is the inverse of the human one.

Keywords

Cognitive aspects, Configuration knowledge engineering, Large language models, Feature models

1. Introduction

Configuration knowledge engineering (CKE) is concerned with authoring and maintaining the *knowledge bases (KBs)* that drive knowledge-based configuration systems. CKE has matured over four decades, from the rule base of R1/XCON in the early 1980s [1] through model-based configuration [2] to the feature models that drive *software product lines (SPLs)* [3, 4]. To develop and maintain such KBs, the field has produced solver-based tooling that spans their lifecycle, from model-based diagnosis [5, 6, 7], testing and debugging [8, 9, 10], and redundancy detection [11, 12] during construction to analysis operations for validation and maintenance [13, 14]. Although this tooling improves KB development, the principal bottleneck lies upstream, in the human authoring of these KBs. This process is inherently error-prone: cognitive overload, incomplete domain knowledge, and outdated rules lead engineers to encode faulty, redundant, or inconsistent constraints, and the way the same knowledge is represented directly shapes the cognitive effort involved [15, 16].

Among these causes, how knowledge engineers represent and formalize constraints has itself been studied empirically as a source of such errors in CKE [15, 17, 18]. Two of these findings motivate this paper. Felfernig et al. [17] report that human knowledge engineers solving a small *constraint satisfaction problem (CSP)* [19] make 21% errors on the implication form $A \rightarrow B$ and 96% errors on its logically equivalent contrapositive $\neg B \rightarrow \neg A$. Felfernig et al. [18] report that engineers translating

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany
v.m.le@tugraz.at (V. Le); sebastian.lubos@tugraz.at (S. Lubos); trang.tran@tugraz.at (T. N. T. Tran); alexander.felfernig@tugraz.at (A. Felfernig)
0000-0001-5778-975X (V. Le); 0000-0002-5024-3786 (S. Lubos); 0000-0002-3550-8352 (T. N. T. Tran); 0000-0003-0108-3146 (A. Felfernig)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

natural-language (NL) statements into formal encodings mistake directional "only compatible with" relations for biconditionals in 60% of cases, and that none identify the redundant constraints in a small CSP (0% exact match). This latter finding connects to the linguistic-ambiguity literature in requirements engineering [20] and motivated automated redundancy detection algorithms such as CoreDiag [12]. Together these studies identify four cognitive dimensions along which human performance in CKE degrades: *representation choice*, *constraint ordering*, *NL-to-formal translation*, and *redundancy detection*.

These results raise a question of *transfer*. *Large Language Models (LLMs)* are now being considered as analysis assistants for CKE, with SPLs as a leading use case. With appropriate prompting techniques [21, 22], LLMs map NL into formal artefacts such as CSPs [23], answer-set programs [24], and *feature models (FMs)* [25], with benchmarks confirming both the capability and its limits [26]. In this context, Le et al. [27] show that frontier LLMs execute the sixteen standard analysis operations [13] on FMs with accuracy approaching the solver-based oracle [28]. The FMs in their study are expressed as *blueprints*, semi-formal NL descriptions of feature hierarchies and constraints, such as *"a Smartwatch must have a Connector and a Screen"* or *"the Connector can be Cellular, Wifi, or both"*. That result establishes *feasibility*, but not *cognitive robustness*, i.e., whether an LLM returns the same answer when the same constraint is rewritten in a logically equivalent form, reordered, or renamed, or when a description is phrased ambiguously. This gap matters in practice. Le et al. [27] report that misinterpreting FM relationships, such as interpreting *or*/*alternative* as *mandatory*, is the most frequent of three failure modes, without identifying its source.

Furthermore, the human error patterns were established in a setting that differs from real LLM-assisted CKE in two respects. First, real feature models are far larger and more structurally complex than the small synthetic CSPs of the human studies, so what counts in practice is whether the effects hold at that scale. Second, real feature models carry meaningful names rather than domain-independent variables, so an LLM may bring world knowledge to bear that could either aid or mislead it. Both differences bear directly on whether LLMs can be trusted as authoring assistants, and neither can be settled from the human studies alone.

This paper analyses the impact of different constraint representations on the knowledge acquisition behaviour of LLMs along the four human-derived dimensions (*representation*, *ordering*, *formalization*, *redundancy*), plus a fifth dimension novel to LLMs: *naming*. We organise the experimental design around three research questions. *(RQ-A) Transfer*: do the human cognitive error patterns of [17, 18] along the four dimensions of constraint representation, ordering, formalization, and redundancy replicate when LLMs are the subjects? *(RQ-B) Generalisation to FM*: do those effects persist when stimuli move from small synthetic CSPs to inputs derived from real, industrial-scale feature models, namely *Universal Variability Language* (UVL) models, FM-derived CSPs, and NL blueprint fragments? *(RQ-C) World-knowledge invocation*: do LLMs invoke pretrained world knowledge when feature names carry domain meaning, and does the invocation help or hurt accuracy on configuration tasks?

In this paper we make three contributions. First, we design an experimental framework that recasts the four human cognitive dimensions [17, 18] as LLM-testable manipulations, extends them from small synthetic CSPs to FM-derived inputs at industrial scale, adds a fifth *naming* manipulation, and comprises nine experimental settings evaluated across five LLMs.[1] Second, we show that the human error patterns largely do not transfer to LLMs. The representation, ordering, and formalization manipulations that derail human knowledge engineers leave the five LLMs at or near ceiling, with a single formalization task as the only exception, and on redundancy detection three of them exceed the human baseline. In the same experiments we identify three LLM-specific vulnerabilities: (1) one LLM collapses as the feature model grows large when asked to produce a valid configuration, while the other four remain accurate; (2) abstract feature identifiers (`f1`...`fN`) outperform domain-meaningful feature names across two distinct task types; and (3) all five LLMs fail to identify the *mandatory* relationship under ambiguous NL descriptions and conflate it with cross-tree *requires* and *or* relationship, which locates a concrete source of the semantic-slip failure mode reported by [27]. These results indicate a cognitive-robustness

[1] Code, stimuli, raw model outputs, evaluation results, and analysis scripts are available at https://github.com/AIG-ist-tugraz/cognitive-robustness-cke/releases/tag/v1.0-confws2026.

profile inverse to the human one. Third, we derive basic recommendations for the design of LLM-assisted CKE pipelines. Together, these contributions carry a practical message for CKE. LLMs are unaffected by the manipulations that derail human knowledge engineers. This robustness motivates their use as assistants in the human authoring stage. This assistance is safe only once the three LLM-specific vulnerabilities we identify are mitigated.

The remainder of the paper is organised as follows. Section 2 introduces feature models, UVL, and the analysis tasks. Section 3 reviews the cognitive-aspects studies and prior work on LLMs in CKE. Section 4 describes the experimental design. Sections 5, 6, and 7 report the results for each RQ. Section 8 discusses cross-experiment patterns and implications for LLM-assisted CKE pipelines. Section 9 discusses threats to validity, and Section 10 concludes with recommendations and future work.

2. Background

This section introduces the artefacts the experiments operate on and the tasks they perform over them. It moves from feature models and their textual form in UVL to the NL blueprints that describe them, then to the CSPs they translate into and the reasoning and analysis tasks defined over them. The Smartwatch feature model of Figure 1 runs as an example throughout.

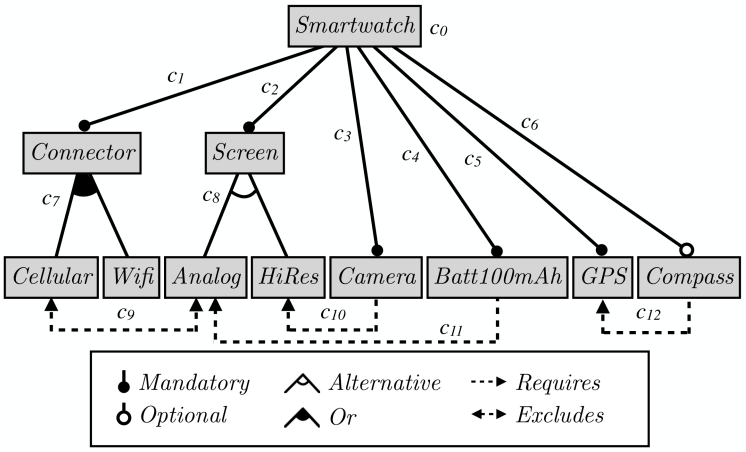
Feature models. Feature models, introduced in the *Feature-Oriented Domain Analysis (FODA)* method of Kang et al. [4], have become the de facto representation of variability and commonality in an SPL [3]. An FM is a tree of features connected by structural relationships, together with cross-tree constraints [13]. The four structural relationships are *mandatory* (the child is included whenever the parent is), *optional* (the child may or may not be included), *or* (if the parent is included, at least one child of the group is selected), and *alternative* (if the parent is included, exactly one child of the group is selected). The two cross-tree constraints are *requires* (if one feature is present, another must also be present) and *excludes* (two features are mutually exclusive). A *configuration* is a selection of features that satisfies every relationship and every cross-tree constraint.

For demonstration, we use a simplified and intentionally faulty Smartwatch feature model (SMW) as a *running example* throughout the paper. Figure 1 shows it in four representations (feature model diagram, UVL, blueprint, and FM-derived CSP): a *Smartwatch* must have a *Connector*, a *Screen*, a *Camera*, a *Batt100mAh* (a 100 mAh battery), and a *GPS*, with an optional *Compass*. The *Connector* is one or both of *Cellular* and *Wifi*, and the *Screen* is either *Analog* or *HiRes* (high resolution). The four cross-tree constraints are: *Analog* excludes *Cellular*, *Camera* requires *HiRes*, *Batt100mAh* requires *Analog*, and *Compass* requires *GPS*. By design, *Camera requires HiRes* and *Batt100mAh requires Analog* jointly force both *HiRes* and *Analog*, which the *alternative* Screen forbids. Because of the conflict, no valid configuration exists for this feature model.

UVL. UVL [29] has emerged as a unified textual language for feature modeling, and we adopt it as the textual syntax throughout this paper. It encodes the four structural relationships as keyword-tagged child groups and the cross-tree constraints as Boolean expressions, as Figure 1b shows for the running example. We use only its Boolean level, not the arithmetic or type extensions.

Blueprints. Le et al. [27] introduced *blueprints*, an NL representation of feature models that serves as the input to LLM-based analysis operations. A blueprint is a semi-formal NL description of a model's features and dependencies, such as *"a Smartwatch must have a Connector and a Screen"* or *"a Camera requires a HiRes Screen"*. Figure 1c gives the blueprint of the running example. Both the UVL form and the blueprint form are used as LLM inputs across the experiments in this paper.

Constraint satisfaction and configuration. A feature model can be transformed into a formal representation, such as a CSP, by mapping each feature to a Boolean variable (selected or not selected) and by translating both the hierarchical relationships and the cross-tree constraints into logical constraints.



(a) Feature model diagram

```
features
    Smartwatch
        mandatory
            Connector
                or
                    Cellular
                    Wifi
            Screen
                alternative
                    Analog
                    HiRes
            Camera
            Batt100mAh
            GPS
        optional
            Compass

constraints
    !(Analog & Cellular)
    Camera => HiRes
    Batt100mAh => Analog
    Compass => GPS
```

(b) UVL

- A [Smartwatch] must have at least one type of [Connector], a [Screen], a [Camera], a [Batt100mAh], and a [GPS].
- The [Connector] can be [Cellular], [Wifi], or any combination of them.
- The [Screen] can be [Analog] or [HiRes].
- The [Smartwatch] may also include a [Compass].
- [Analog] and [Cellular] exclude each other.
- A [Camera] requires a [HiRes].
- A [Batt100mAh] requires an [Analog].
- A [Compass] requires a [GPS].

(c) Blueprint

c_0	$Smartwatch = t$
c_1	$Smartwatch \leftrightarrow Connector$
c_2	$Smartwatch \leftrightarrow Screen$
c_3	$Smartwatch \leftrightarrow Camera$
c_4	$Smartwatch \leftrightarrow Batt100mAh$
c_5	$Smartwatch \leftrightarrow GPS$
c_6	$Compass \rightarrow Smartwatch$
c_7	$Connector \leftrightarrow or(Cellular, Wifi)$
c_8	$Screen \leftrightarrow xor(Analog, HiRes)$
c_9	$\neg(Analog \wedge Cellular)$
c_{10}	$Camera \rightarrow HiRes$
c_{11}	$Batt100mAh \rightarrow Analog$
c_{12}	$Compass \rightarrow GPS$

(d) FM-derived CSP

Figure 1: The Smartwatch running example in four representations of the same feature model: (a) the feature model diagram, (b) the UVL text, (c) the NL blueprint, and (d) the FM-derived CSP.

A complete set of rules for this translation is given by Benavides et al. [13]. Formally, a CSP is a tuple (V, D, C) with variables V, finite domains D, and constraints C, and a *solution* is an assignment of values from D to the variables of V that satisfies every constraint in C [19]. A *configuration task* is the special case $C = C_{KB} \cup REQ$, where C_{KB} is a *configuration knowledge base* that encodes the product model and REQ collects user requirements, and a *configuration* is a solution of the configuration task. Figure 1d shows the FM-derived CSP of the running example, where $c_0 : Smartwatch = t$ (t denotes true) is a *root constraint*, avoiding the derivation of empty configurations. This transformation enables identifying configurations, performing solver-based reasoning, and executing analysis operations on FMs.

Conflict set identification. When a configuration task has no solution, we are interested in explanations as to why no solution could be identified. *Minimal conflict sets*, also denoted as *minimal unsatisfiable subsets* (MUS) [30], are a common means of explaining such inconsistent situations. Formally, a minimal conflict set CS is a minimal set of *possibly faulty* constraints that is inconsistent together with the *background knowledge* C_{KB}, where minimal means no proper subset $CS' \subset CS$ is inconsistent with C_{KB} [6]. In the running example (Figure 1d), with the feature hierarchy (c_0 to c_8) as the background knowledge C_{KB} and the cross-tree constraints (c_9 to c_{12}) as possibly faulty, $\{c_{10}, c_{11}\}$ is a minimal conflict set: because *Camera* and *Batt100mAh* are mandatory, c_{10} entails *HiRes* and c_{11} entails

Table 1
The six task types used across experiments and the abbreviations used as labels throughout the paper.

Task	Abbreviation
Find a configuration	*find-config*
Find a minimal conflict set	*find-conflict*
Formalization	*formal*
Redundancy detection	*red-detect*
Satisfiability check	*sat*
Configuration counting	*count*

Analog, but the *alternative* Screen allows only one of them, so no configuration exists.

Redundancy detection. A constraint is *redundant* when removing it leaves the solution space unchanged, and redundancy detection is the task of flagging such constraints, which neither restrict nor enlarge the set of valid configurations [12]. In the running example (Figure 1d), once c_{11} is removed so that the model has valid configurations, c_{12} (*Compass requires GPS*) is redundant: *GPS* is mandatory, so the implication always holds, and removing c_{12} leaves the set of configurations unchanged. Although semantics-preserving, redundant constraints reduce model understandability and inflate maintenance and solver cost, which makes their reliable detection a practical concern [31].

Analysis operations. Conflict and redundancy detection are part of the catalogue of *analysis operations* of [13], which formalises the standard reasoning tasks over FMs. Two further such operations are a *satisfiability check*, which decides whether an FM has at least one valid configuration, and *configuration counting*, which computes the number of valid configurations. In the running example (Figure 1d), the satisfiability check returns false (the model is void) and the configuration count is zero. The solver-based oracle FLAMA [28] executes these operations on UVL inputs.

Tasks. Six task types recur across the experiments. Five appear above, namely *find-config* (a valid configuration), *find-conflict* (a minimal conflict set), *red-detect* (the redundant constraints), and *sat* and *count* (the satisfiability check and configuration counting). The sixth, *formalization* (*formal*), is a translation task in which the model is given an NL statement and must select its correct formal encoding from a set of candidate constraints. Table 1 lists all six with the abbreviations used as labels throughout the paper.

3. Cognitive Aspects of Configuration Knowledge Engineering: Studies and Replication Template

The cognitive aspects of CKE have been studied empirically since Felfernig et al. [15]. That work showed that different expressive means for the same chunk of knowledge trigger different cognitive overheads on the knowledge engineer, and that the choice of representation significantly affects knowledge-base acquisition and maintenance. Two studies extend this line directly and supply the methodological template that our experiments replicate.

Felfernig et al. [17] ran two within-subject studies on small synthetic CSPs (3 to 10 variables, 5 to 15 mixed constraints) over domain-independent variables $v_1, v_2, \ldots$ *Study A* varied the order of the constraints on two synthetic CSPs across three orderings, namely variable-similarity (constraints that share variables placed together), operator-similarity (constraints with the same operators placed together), and random. In each ordering, subjects were asked to find a solution or a minimal conflict, and variable-similarity ordering halved the error rate of the random baseline. *Study B* varied the form of an implication on two further CSPs across five logically equivalent encodings, $A \rightarrow B$ (R_1), $\neg A \vee B$ (R_2), $\neg B \rightarrow \neg A$ (R_3), $\neg(A \wedge \neg B)$ (R_4), and $B \leftarrow A$ (R_5), and asked subjects to find a solution. The contrapositive

form R_3 produced 96% error against 21% for the implication form R_1, a 75 percentage point (pp) gap on a logically identical constraint.

Felfernig et al. [18] extended this line from representation choice to formalization, addressing what the authors call a neglected area of CKE research, namely the cognitive understandability of transforming informally specified knowledge into a formal representation. Ten translation tasks in a financial-advisory domain, over the variables *return rate* (*rr*), *willingness to take risks* (*wr*), and *investment period* (*ip*), were given to ten knowledge engineers, each asking for the formal encoding of an NL statement. Four concern compatibility and incompatibility relations. *Task 1* encodes *"a low willingness to take risks is incompatible with a high return rate"* as either a compatibility or an incompatibility set. *Task 6* encodes the directional *"a high return rate is only compatible with a high willingness to take risks"*, which must not collapse into an equivalence. *Task 7* encodes *"a shortterm investment period is incompatible with a high return rate"* as either $\neg(ip{=}short \wedge rr{=}high)$ or $ip{=}short \rightarrow \neg\, rr{=}high$. *Task 9* encodes an incompatibility whose two directions are stated explicitly. Three concern *requires* constraints. *Task 2* shares a left-hand side, as in *"a high return rate requires a high willingness to take risks and a longterm investment period"* ($rr{=}high \rightarrow (wr{=}high \wedge ip{=}long)$). *Task 3* shares a right-hand side, as in *"a shortterm period and a high return rate both require a high willingness to take risks"*. *Task 4* has a disjunctive right-hand side, as in *"a high return rate requires a longterm investment period or a high willingness to take risks"* ($rr{=}high \rightarrow (ip{=}long \vee wr{=}high)$). The remaining three are *Task 8*, combined properties through enumerated negation (*"a return rate that is not low and not medium requires an investment period that is not shortterm and not mediumterm"*, which merely restates *a high return rate requires a longterm investment period*); *Task 5*, spotting the two redundant constraints hidden among eleven over ten variables (for instance $v_8 \neq v_9$ and $v_9 \neq v_8$, both entailed by $v_9 > v_8$); and *Task 10*, describing a whole feature model in prose, where the *or* relationship is the hardest element to convey unambiguously. Two findings of this paper underpin our experimental design. First, directional "only compatible with" relations were misread as biconditional in 60% of cases, while the same logical content with the explicit phrase "and vice-versa" was read correctly in 90% of cases. Second, no participant identified the two redundant constraints in a CSP of eleven constraints, and the participants who returned faulty answers spent less time than those who returned correct answers, indicating that confidence on this task is uncorrelated with correctness. This second finding directly motivated the automated redundancy detection algorithm *CoreDiag* [12]. The ambiguity findings of [18] also connect to the long-standing literature on linguistic ambiguity in requirements specifications [20].

These two studies establish that constraint representation choice, constraint order, NL-to-formal translation, and redundancy detection are four distinct cognitive dimensions along which the CKE process degrades. They form the four-dimension template that our experiments replicate with LLMs as subjects. We reuse the *Study A*/*Study B* and *Task n* labels of [17, 18] throughout the paper to refer back to these source experiments.

Machine learning has been applied across feature modeling, configuration, and constraint solving [32, 33]. More recently, LLMs have been used to map NL descriptions onto formal artefacts such as CSPs [23, 34], answer-set programs [24], and feature models [25]. Le et al. [27] propose LLMs as an engine for the early validation of FMs, targeting SPL scoping [35], the early phase of SPL engineering where variability decisions are first articulated. Their benchmark evaluates twelve LLMs on the sixteen analysis operations [13] applied to ten FMs ranging from six to nearly seven thousand features, presented as blueprints (Section 2) rather than as formal UVL. The top three reach 88 to 90% average accuracy against the oracle FLAMA [28]. The study reports three failure modes, namely *semantic slips* on FM relationships, *incomplete propagation* on counting operations, and *context-window exhaustion* on the largest FMs. Semantic slips are by far the most frequent of the three, and are squarely a cognitive-aspects phenomenon, where an LLM misreads, for example, an *or* or *alternative* group as *mandatory*, the same kind of representation-induced error that the human studies above document. Le et al. [27] report this slip without diagnosing its source, an open question that our study takes up.

Together, these foundations set up our study. The cognitive-aspects studies of [17, 18] supply the methodological template that our experiments replicate, while the blueprint-based pipeline of [27] serves both as proof of feasibility, since it shows that LLMs can execute the analysis operations at all,

Table 2
Nine experimental settings grouped by research question. FM acronyms follow [27] Table 2: SMW = Smartwatch, COM = Computer, SEA = Subsea Control System, CVE = Cybersecurity Vulnerability, BDB = Berkeley DB. See Table 3 for feature model characteristics. The *Task* column uses the abbreviations defined in Table 1; the *Task n* labels in the *Experiment input* column follow [18] (Section 3). Each RQ-B experiment (E1b–E4b) extends the RQ-A experiment of the same axis; E5 is a novel naming experiment.

Exp.	Axis	Task	Experiment input
RQ-A (Transfer): do human cognitive patterns replicate on LLMs?			
E1a	Representation	*find-config*	Synthetic CSPs of *Study B* [17]
E2a	Formalization	*formal*	NL → CSP (*Tasks 1–4, 6–9*) [18]
E3a	Redundancy	*red-detect*	CSP of 11 constraints (*Task 5*) [18]
E4a	Ordering	*find-config*, *find-conflict*	Synthetic CSPs of *Study A* [17]
RQ-B (FM Generalisation): do those effects persist on FM-derived stimuli?			
E1b	Representation	*find-config*	UVL FMs (SMW, COM, SEA)
E2b	Formalization	*formal*	Blueprint fragments [27] from SMW, COM, SEA, CVE
E3b	Redundancy	*red-detect*	UVL FMs (COM, SEA, CVE, BDB)
E4b	Ordering	*find-config*, *find-conflict*	FM-derived CSP (SMW, COM, SEA)
RQ-C (World-knowledge invocation): do LLMs invoke world knowledge encoded in feature names, and does it help or hurt?			
E5	Naming	*find-config*, *sat*, *count*	UVL FMs (COM, SEA) × 3 naming variants (*abstract*, *domain-aligned*, *domain-scrambled*)

and as a source of the failure modes we investigate. The next section turns these foundations into a concrete experimental design.

4. Experimental Design

This section describes the experimental design, organised around the three research questions (Section 1) and summarised in Table 2. *RQ-A* (*Transfer*) is addressed by the four synthetic-CSP replications E1a–E4a, which test whether the human error patterns of [17, 18] reappear when LLMs are the subjects. *RQ-B* (*Generalisation to FM*) is addressed by the four FM-derived extensions E1b–E4b, which lift each manipulation from synthetic CSPs to industrial-scale feature models. *RQ-C* (*World-knowledge invocation*) is addressed by the naming experiment E5, which tests whether the domain meaning of feature names helps or hurts LLM reasoning. Each experiment is detailed in its own subsection of Sections 5–7, while the rest of this section describes the setup shared across them. Several experiments attach a *foundational-knowledge scaffolding* to the prompt: a set of reference documents (detailed below) that supply the background needed to interpret UVL feature models. Three ablation variants of this scaffolding, each removing one or more of those documents, are applied to each of E1b and the *find-config* task of E5, giving four conditions per experiment including the full-scaffolding baseline (Section 6.1).

Experiment inputs. The experiments take two kinds of input: synthetic CSPs and FMs.

Constraint satisfaction problems. Synthetic CSPs from [17, 18] serve as inputs for E1a, E3a, and E4a, kept at the original-paper scale (3–10 variables, 5–15 mixed constraints) to enable direct comparison with the human baselines of those studies. The NL-to-formal translation candidates of E2a are CSP fragments from [18] at the same scale. E4b also derives CSPs from UVL FMs by applying the FM-to-CSP translation rules of [13].

Feature models. Five feature models from the dataset of [27] populate the FM-derived experiments. Across these experiments, a model appears in one of three forms, namely a UVL FM (E1b, E3b, E5), an NL blueprint fragment (E2b), or an FM-derived CSP (E4b). Table 3 reports their structural metrics.

Table 3
Feature models used in this paper. Structural metrics from [27] Table 2. "#Rel." is the count of structural relationships (*mandatory*, *optional*, *or*, *alternative*). "#CTC" is the count of cross-tree constraints (*requires*, *excludes*).

FM	Full name	#Feat.	#Rel.	#CTC	Depth	Used in
SMW	Smartwatch	13	8	2	2	E1b, E2b, E4b
COM	Computer	48	25	21	3	E1b, E2b, E3b, E4b, E5
SEA	Subsea Control System	145	73	13	10	E1b, E2b, E3b, E4b, E5
CVE	Cybersecurity Vulnerability	169	15	153	4	E2b, E3b
BDB	Berkeley DB	117	54	282	5	E3b

Table 4
LLMs used in this paper, ordered by decreasing average accuracy on the benchmark of [27] (89.7%, 88.9%, 88.2%, 78.3%, and 68.0% respectively). Provider model IDs link to the corresponding model documentation. Reasoning effort follows each provider's API documentation. The Grok 4 Fast Reasoning model benchmarked by [27] has been retired since publication. Following Grok's compatibility guidance, we substitute grok-4.3 with low reasoning effort as the closest match.

Short name	Provider model ID	Type	Reasoning effort
Grok	grok-4.3 [36]	Reasoning-optimized	low
GPT5	gpt-5-mini [37]	Reasoning-optimized	high
Gemini	gemini-2.5-pro [38]	Reasoning-optimized	default
Claude	claude-sonnet-4-6 [39]	Non-reasoning	—
GPT4.1	gpt-4.1-mini [37]	Non-reasoning	—

The five FMs span a broad range of size and constraint density. SMW is the smallest, comparable in scale to the synthetic CSPs of E1a. COM (48 features) and SEA (145 features) are just below and just above 100 features, the size at which the scale-cliff of E4b emerges. CVE and BDB have far higher ratios between cross-tree constraints and features than the others, so they serve as the inputs for the redundancy detection test of E3b across its six redundancy types. E1b, E3b, and E4b use the abstract variants distributed with [27] (feature identifiers replaced by `f1`...`fN`). The short identifiers reduce token counts on the larger FMs (CVE, BDB) and keep each prompt well within the context window of every LLM. SMW is the exception. It is small enough not to need shortening and no abstract variant is distributed for it, so E1b and E4b use it with its original feature names. E5 uses the original feature names because naming is the experimental variable.

LLM selection. Table 4 lists the five LLMs tested. The selection follows the ranking of [27], who find three reasoning-optimized models at the top (Grok 4 Fast Reasoning, GPT-5 mini, Gemini 2.5 Pro). We adopt these three as the reasoning-model contingent, paired with the two highest-ranked non-reasoning models (Claude Sonnet 4.6 and GPT-4.1 mini). Throughout the paper we use the short names *Grok*, *GPT5*, *Gemini*, *Claude*, and *GPT4.1*, in the same decreasing-accuracy order. Coverage is not exhaustive. Section 9 discusses generalisation.

Evaluation pipeline. Every experiment uses the same three-part prompt pipeline, illustrated in Figure 2 on the SMW *find-config* task. A *system prompt* fixes the model role and attaches the foundational-knowledge scaffolding, a *user prompt* carries the experiment input (here the SMW UVL of Figure 1) and the task instruction, and an *output contract* requires the model to reason inside `<reasoning>` tags and emit its answer inside `<answer>` tags, which an eval script parses and scores against the ground truth. This mirrors the three-part pipeline of [27], except that our inputs are UVL feature models rather than blueprints, so the foundational-knowledge scaffolding adds a UVL reference and grammar (detailed below).

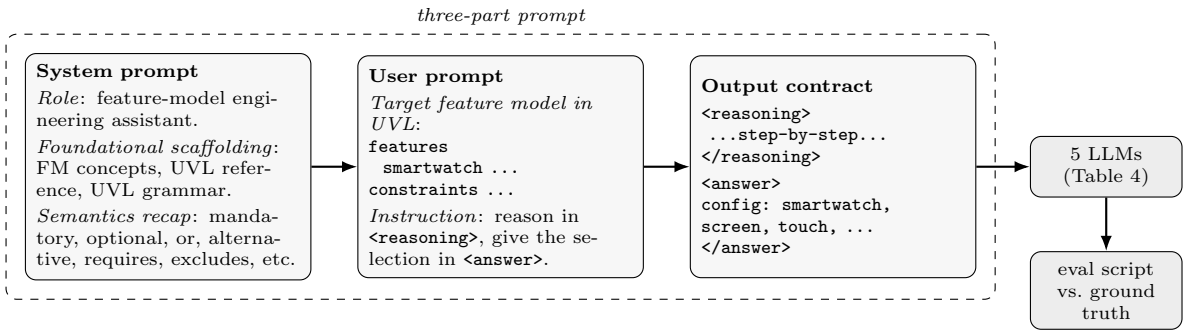


Figure 2: The three-part prompt pipeline (system prompt, user prompt, output contract), shown on the SMW *find-config* task. Each prompt is sent to the five LLMs of Table 4. An eval script parses the `<answer>` block and scores it against the ground truth. Cf. Figure 1 of [27].

Foundational-knowledge scaffolding. The prompts for E1b, E3b, and the *find-config* task of E5 attach three reference documents to the system message (Figure 2): a brief on feature model concepts, a UVL reference, and the UVL grammar. They act as foundational-knowledge *scaffolding*, supplying the background needed to interpret UVL feature models and controlling for cross-provider variance in how much UVL each provider's training corpus contained. The *sat* and *count* tasks of E5 build on the system prompts of [27] behind their A010 (satisfiability) and A012 (configuration counting) operations, extended with the same scaffolding. Section 6.1 reports the ablation that removes individual scaffolding components and quantifies their contribution.

Sample size and statistical framing. Each (experiment, condition, LLM) cell is run $n = 3$ times. With so few runs in each cell, the paper relies throughout on *aggregated patterns*: totals for each LLM (typically 15–36 observations per condition, after aggregating over the multiple stimuli in each condition), monotonic gradients across FM size or scaffolding variant, and side-by-side comparison with the human baselines of [17, 18]. Section 9 discusses the confidence interval for each cell.

Experiment-input conventions. Three conventions address potential biases in the experimental measurements. (i) Variable identifiers in the replication inputs from [17, 18] are renamed (`wr/rr/ip` → `x1/x2/x3` in E2a, v → x in E3a) to mitigate verbatim training-data contamination. (ii) In the multiple-choice experiments (E2a and E2b), candidate answers are shuffled per (input, LLM, run) with a deterministic seed for each run to control for positional bias. The recorded permutation maps the chosen position back to the canonical answer. (iii) UVL FM inputs are presented in full, with hierarchy, groups, and cross-tree constraints intact (E1b, E3b, and the *find-config* task of E5), to avoid partial-input effects.

5. Transfer of Human Patterns (RQ-A)

In this section, we report the four synthetic-CSP experiments that test whether human cognitive error patterns documented by [17, 18] replicate when LLMs are the subjects. Each subsection pairs with the corresponding FM-scale extension in Section 6.

5.1. E1a: Representation Effect on Synthetic CSP

Study B of [17] documented that humans make 21% errors on the implication form $A \rightarrow B$ and 96% errors on its logically equivalent contrapositive $\neg B \rightarrow \neg A$. The two forms carry the same logical content but produce very different human error rates. E1a asks whether LLMs share this human-specific difficulty to constraint form. The experiment replicates the original design: five logically equivalent encodings of the same constraint set, $A \rightarrow B$ (R_1), $\neg A \vee B$ (R_2), $\neg B \rightarrow \neg A$ (R_3), $\neg(A \wedge \neg B)$ (R_4), and $B \leftarrow A$ (R_5),

Table 5
E1a accuracy by form. All five LLMs ceiling on both CSPs across the five forms. The human baseline for R1–R5 from [17] Table 8 is shown for reference.

CSP	Form	Grok	GPT5	Gemini	Claude	GPT4.1	Human
1	R1 ($A \rightarrow B$)	1.00	1.00	1.00	1.00	1.00	0.79
1	R2 ($\neg A \vee B$)	1.00	1.00	1.00	1.00	1.00	0.50
1	R3 ($\neg B \rightarrow \neg A$)	1.00	1.00	1.00	1.00	1.00	0.04
1	R4 ($\neg(A \wedge \neg B)$)	1.00	1.00	1.00	1.00	1.00	0.27
1	R5 ($B \leftarrow A$)	1.00	1.00	1.00	1.00	1.00	0.75
2	R1...R5	1.00	1.00	1.00	1.00	1.00	0.50–0.86

applied to two CSPs, one with 5 variables and 7 requires constraints and one with 3 variables and 5 incompatibility constraints. For each (CSP, form) cell, the LLM is asked to find a configuration that satisfies the constraints (*find-config*, Table 1). All five LLMs answered correctly across the design space (Table 5). For comparison, the human R3 contrapositive form produced 96% error in the original. The LLM/human gap is therefore 96 pp at this scale, near the theoretical maximum. We interpret this null result, i.e., the absence of any form-related accuracy effect, as a methodological baseline rather than a counter-finding to [17]. These inputs are too small to expose such an effect on our LLMs, including on R3. The question of whether the effect re-emerges at FM scale is the subject of E1b.

5.2. E2a: Formalization through Translation Tasks

Felfernig et al. [18] documented systematic mistranslations when humans encode NL statements as formal CSP, with task-specific human error rates up to 100% on enumerated-negation forms (*Task 8*) and 60% on directional "only compatible with" forms (*Task 6*). E2a asks whether LLMs make the same mistranslations. The experiment replicates eight of the original translation tasks (*Tasks 1–4* and *6–9*), including the directional "only compatible with" stimulus (*Task 6*). For each task, the LLM is presented with an NL statement and asked to pick the correct formal encoding from a list of candidates (*formal*, Table 1). All LLMs except GPT4.1 score 100% on seven of the eight tasks. The exception is Task 1 (compatibility vs. incompatibility), where only GPT5 stays at 100% while the others drop to 33–67%, with every error a partial mislabelling of one candidate encoding rather than a full miss. Averaged across the eight tasks, GPT4.1 scores 87.5% and the other four exceed 90%. As a contamination check, we re-ran the original (un-renamed) wording of four high-impact tasks (*4, 6, 8, 9*) on two LLMs. Accuracy matched the renamed version exactly (0 pp), confirming the results are not inflated by memorisation of the original tasks.

5.3. E3a: Redundancy Detection on Synthetic CSP

Task 5 of [18] documented 0% human exact-match on identifying the two redundant constraints in a CSP of eleven constraints. E3a asks whether LLMs can do better than this 0% human result. The experiment presents the same input to the five LLMs, which are asked to identify the redundant constraints (*red-detect*, Table 1). Exact-match rates vary widely across the LLMs: Claude 100% (3/3), Grok 67% (2/3), GPT4.1 33% (1/3), Gemini and GPT5 0% exact-match but partial-correct on all 3 runs (one of the two redundancies identified). The human baseline of [18] is 0% exact-match. Three of the five LLMs exceed the human baseline (Table 6). The remaining two match it at 0% exact-match but score partial-correct.

5.4. E4a: Ordering Effect on Synthetic CSP

Study A of [17] documented that variable-similarity ordering of constraints yields lower human error rates than operator-similarity or random orderings. E4a asks whether LLMs are sensitive to constraint order. The experiment replicates the original design on two synthetic CSPs, one with 5 variables and

Table 6
E3a redundancy detection accuracy on the input of the redundancy task of [18]. "Exact" counts runs that returned exactly $\{c_0, c_5\}$. "Partial" counts runs that returned a proper subset or superset.

LLM	Exact (3 runs)	Partial	Wrong	Exact rate
Grok	2	1	0	0.67
GPT5	0	3	0	0.00
Gemini	0	3	0	0.00
Claude	3	0	0	1.00
GPT4.1	1	2	0	0.33
Human (2015)	0	n/a	n/a	0.00

Table 7
The four UVL-expressible forms of a *requires* cross-tree constraint $A \rightarrow B$, used as the form manipulation in E1b, illustrated on *Camera requires HiRes*. The reverse-implication form R_5 ($B \leftarrow A$) of [17] is not expressible in UVL and is omitted.

Form	Logical	UVL
R_1 implication	$A \rightarrow B$	`Camera => HiRes`
R_2 disjunction	$\neg A \vee B$	`!Camera | HiRes`
R_3 contrapositive	$\neg B \rightarrow \neg A$	`!HiRes => !Camera`
R_4 negated conjunction	$\neg(A \wedge \neg B)$	`!(Camera & !HiRes)`

15 mixed constraints and one with 10 variables and 10 constraints, under three orderings (variable-similarity, operator-similarity, random) and two variants per CSP (one satisfiable, one unsatisfiable). For satisfiable variants the LLM is asked to find a configuration (*find-config*, Table 1). For unsatisfiable variants it is asked to find a minimal conflict set (*find-conflict*). All five LLMs answered correctly across the design space. As with E1a, we frame this as a methodological baseline. The question of whether the ordering effect emerges at FM scale is the subject of E4b (Section 6.4).

6. FM Generalisation (RQ-B)

This section reports the four FM-derived extensions paired with the synthetic-CSP replications of Section 5. Each extension tests whether the effect (or its absence) persists when the experiment input is derived from a real feature model. The experiment input takes one of three forms: a UVL FM (E1b, E3b), an FM-derived CSP (E4b), or an NL blueprint fragment (E2b).

6.1. E1b: Representation Effect on FM Scale

E1a found all five LLMs at ceiling on small synthetic CSPs (3–5 variables, 5–7 constraints). At this scale, contrapositive parsing does not exceed any LLM's capacity, so the human representation effect cannot be tested. E1b asks whether the effect re-emerges when the same form manipulation is applied to UVL FMs at FM scale. The experiment rewrites the cross-tree constraint block of three UVL FMs (SMW, COM, and SEA in Table 3) in each of four forms (Table 7). The UVL syntax does not support reverse implication, so R_5 is dropped. For each (FM, form) cell, the LLM is asked to find a configuration that satisfies the constraints (*find-config*, Table 1).

The representation effect emerges only on the smallest FM (Figure 3, left panel). At baseline scaffolding, SMW shows accuracy ranging from 33% on R3 to 60% on R2 when averaged across LLMs. COM (medium, 21 cross-tree constraints) is near-ceiling across all forms. SEA (large, 13 cross-tree constraints) stays at a uniform 67% plateau. On SMW, the contrapositive R_3 is the hardest form (33%), matching [17], but the full $R_1 > R_2 > R_4 > R_3$ ranking does not reproduce, since R_2 is in fact the easiest form here (60%). Even this R_3-hardest pattern is fragile to the scaffolding ablation reported below. Grok alone shows a divergent pattern: 78% on the disjunctive forms (R2, R4) versus 44% on the implicative forms (R1, R3) at

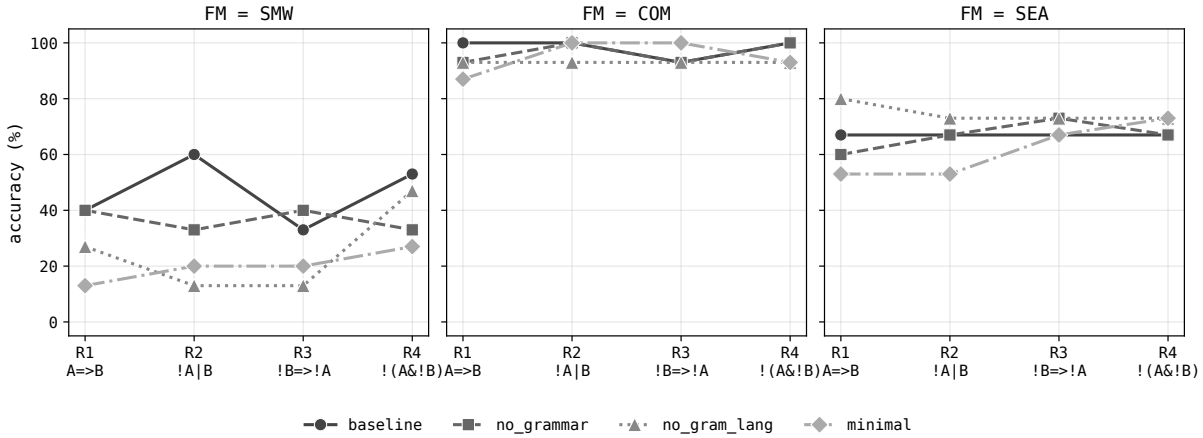


Figure 3: E1b representation effect by FM (panels) and scaffolding condition (lines). The R3-hardest pattern from [17] appears on SMW at baseline only. COM is near-ceiling. SEA is a uniform plateau.

baseline, a 34 pp gap. This signal comes from one LLM and does not generalise across the selection, so we treat it as a candidate for future work rather than a paper finding.

Foundational-scaffolding ablation. Three ablation variants remove, in turn, the UVL grammar reference (`no_grammar`), both the grammar and the UVL language reference (`no_gram_lang`), and all three scaffolding documents (`minimal`). Aggregated across the five LLMs, the `baseline`-to-`minimal` gradient is monotone for the two reasoning-effort models (GPT5 from 92% to 64%, −28 pp, and Gemini from 100% to 86%, −14 pp) and flat to noisy for Claude, GPT4.1, and Grok (Figure 6, left panel). At the `minimal` variant the SMW form ordering scrambles and R_3 is no longer the hardest form. The representation effect is therefore not robust to scaffolding removal. We carry this as a scaffolding-related caveat to the representation-effect finding (Section 9).

6.2. E2b: Relationship and Constraint Identification in Blueprints

Le et al. [27] report that semantic slips on FM relationships and constraints are the most frequent of the three LLM failure modes in their analysis operation pipeline, and they do not diagnose the source. A semantic slip is fundamentally a failure to recover the correct relationship or constraint *type* from its blueprint phrasing. E2b tests this directly. For each such type, we present a blueprint fragment that expresses it, and we ask whether the LLM identifies that type to the exclusion of the readings it is most easily confused with.

We organise the test into four classes by the type of the fragment, i.e., *group* relations (the *or* and *alternative* fragments, the pair most readily conflated), *mandatory* relations, *requires* constraints, and *excludes* constraints. These four classes cover five of the six core relationship and constraint types whose misreading in feature-model blueprints produces the semantic slip. The sixth type, *optional*, enters only as a candidate foil and not as a fragment class, because its "may include" phrasing carries no comparable ambiguity. The fragments and their candidate vocabulary come from the SMW, COM, SEA, and CVE blueprints (Table 3). Each fragment carries four candidate interpretations, exactly one of which is correct. The LLM judges every candidate as a valid reading or not, and a response counts as correct only when it accepts the correct reading and rejects the other three (*formal*, Table 1). This *exclusive-identification* criterion is strict by design, since a response that accepts the correct reading but also marks an incompatible one as valid already commits the semantic slip we study.

The result is uneven across the four classes (Table 8). *Group* relations are identified without error, i.e., every LLM accepts the correct *or* or *alternative* reading and rejects all others (100%). The *or*-versus-*alternative* distinction that a counting-level analysis flags as the strongest confusion site (Appendix A) therefore does not surface at the identification level, which places that counting-level confusion in the enumeration step rather than in the reading of the relationship. *Requires* is also identified well (93%).

Table 8
E2b exclusive-identification accuracy by class. Group relations and *requires* are identified without error, whereas the mandatory relationship is the failure mode.

Class	Grok	GPT5	Gemini	Claude	GPT4.1
Group (*or*/*alternative*)	1.00	1.00	1.00	1.00	1.00
Mandatory	0.00	0.17	0.00	0.00	0.00
Requires	1.00	1.00	0.83	1.00	0.83
Excludes	0.17	0.33	0.00	0.50	0.83

Table 9
Readings the five LLMs accept for the mandatory "must have a [X]" fragment, averaged over runs. The correct parent–child reading is accepted less often than the cross-tree *requires* reading, and only *optional* is rejected outright.

Reading accepted for "must have a [X]"	Acceptance
Cross-tree *requires*	90%
Parent–child *mandatory* (correct)	77%
or-group	60%
alternative-group (SEA stimulus)	47%
optional "may include"	0%

Excludes drops to 37%, almost entirely because the LLMs accept a one-way $A \rightarrow \neg B$ encoding (64%) alongside the correct symmetric reading, which is a completeness lapse rather than a type confusion. *Mandatory* is the failure mode, at 3.3% exclusive identification and the lowest of the four. The mandatory stimulus shows the mechanism (Listing 1).

Listing 1: The E2b mandatory stimulus.

```
Fragment: a [f1] must have a [f2]
  (1) Mandatory: [f1] must include [f2]          <-- correct
  (2) Or-group: parent [f1], child {[f2]}
  (3) Cross-tree: [f1] requires [f2]
  (4) Alternative group: parent [f1], child {[f2]}
```

The mandatory result has a specific structure (Table 9). On the fragment "must have a [X]" the LLMs accept the correct parent–child *mandatory* reading in 77% of runs, but they also accept the cross-tree *requires* reading (90%) and the *or*-group reading (60%), and on the SEA stimulus the *alternative*-group reading (47%). They reject only the *optional* "may include" reading (0%). The same LLMs reject every cross-tree and hierarchy foil on the group fragments (the group class at 100%), so this behaviour is specific to the mandatory phrasing and is not a generic tendency to over-accept. The diagnosis is therefore sharper than a single confusion pair. The blueprint phrase "must have a [X]" does not let the LLMs separate a parent–child mandate from a cross-tree *requires* or a group membership, although they separate it clearly from *optional*. This result locates a concrete and actionable source of the semantic-slip failure mode of [27]. Blueprint authoring should phrase a mandatory child so that it cannot be read as a cross-tree dependency or as a group.

6.3. E3b: Redundancy Detection by Type on Full UVL FMs

E3a found three of five LLMs exceeded the 0% human exact-match baseline on the small stimulus of 11 constraints. E3b asks whether LLMs sustain this redundancy detection advantage at FM scale and across a typed redundancy taxonomy. The experiment extends E3a to four UVL FMs (COM, SEA, CVE, and BDB in Table 3) and to six redundancy types from the FM-analysis catalogue of [13]. Two are structural: *child* $\rightarrow$ *parent* (a cross-tree implication that the feature tree already entails, e.g., *GPS* $\rightarrow$ *Connector* when *GPS* is a tree child of *Connector*) and *alternative-excludes pair* (a cross-tree excludes between two

Table 10
E3b recall by redundancy type (%) on the redundancy detection task, pooled across COM, SEA, CVE, and BDB. n is the total count of injected redundancies of each redundancy type across the four FMs (3 runs per stimulus). Structural redundancy types are at ceiling for every LLM; transitivity is the hardest cross-tree redundancy type.

Redundancy type	n	Grok	GPT5	Gemini	Claude	GPT4.1
Structural						
child → parent	12	100	100	100	100	100
alt-excludes	9	100	100	100	100	100
Cross-tree						
contrapositive	3	100	100	100	100	67
restatement	12	75	100	100	75	67
weakening	15	93	100	100	100	33
transitivity	3	0	100	100	100	0

siblings in an *alternative* relationship, e.g., *Analog excludes HiRes* when *Analog* and *HiRes* are alternatives under *Screen*). Four are cross-tree. Three of them are redundancies of a base implication $GPS \rightarrow Screen$, namely *restatement* ($\neg GPS \vee Screen$), *weakening* ($GPS \rightarrow Screen \vee Camera$), and *transitivity* (entailed by $GPS \rightarrow Cellular$ together with $Cellular \rightarrow Screen$). The fourth, *contrapositive*, is derived from an excludes constraint, so the base $Analog \rightarrow \neg Cellular$ of the running example yields $Cellular \rightarrow \neg Analog$. For each FM, the LLM is asked to flag all redundant constraints (*red-detect*, Table 1).

Table 10 summarises recall by redundancy type and shows that the redundancy type, not LLM capability, is the binding constraint. All five LLMs achieve 100% recall on both structural types. Contrapositive is at 100% across the LLM selection with the single exception of GPT4.1 (67%). Restatement spans 67% to 100%. Weakening averages 85% (range 33–100%) with GPT4.1 alone accounting for the low end at 33%. Transitivity averages 60% (range 0–100%), with GPT4.1 and Grok at 0%.

Two failure modes appear in the exact-match rate for each FM. Gemini and GPT5 false-positive every run on the no-redundancy control constraints, reducing their exact-match rate to 0% across all four FMs even where their recall by redundancy type is perfect. GPT4.1 under-flags weakening and transitivity, producing a sparse-but-correct flagging that earns partial credit on COM (67% exact) but fails entirely on SEA and BDB. We interpret these as two distinct cognitive failure modes, i.e., an over-flag bias and an under-flag bias, each tied to specific LLMs and not predicted by recall alone.

6.4. E4b: Ordering and Scale on FM-Derived CSPs

E4a found all five LLMs at ceiling on the synthetic 5–10 variable CSPs of [17]. The ordering effect therefore cannot be tested at that scale. E4b asks whether the ordering effect (or any scale-dependent effect) emerges when stimuli scale to FM-derived CSPs with 21–377 constraints. The experiment lifts the ordering manipulation to FM-derived CSPs of SMW, COM, and SEA (Table 3). Following [13], each FM relationship is expanded into one or more implications. A mandatory relationship becomes a bi-implication, an optional relationship and a cross-tree constraint each become a single implication, and group relationships (*or*/*alternative*) become a parent-to-children disjunction plus one child-to-parent implication per child, with *alternative* adding the pairwise exclusions. As in E4a, the LLM is asked to find a configuration on satisfiable variants (*find-config*, Table 1) and to find a minimal conflict set on unsatisfiable variants (*find-conflict*).

The ordering manipulation produces no detectable effect across any FM. The size manipulation shows a scale-cliff at one of the five LLMs (Figure 4). GPT4.1 achieves 100% accuracy on SMW (13 features, 21 constraints), 11% on COM (48 features, 128 constraints), and 0% on SEA (145 features, 377 constraints). The other four LLMs remain at 100% across all three FM sizes. The failure mode on SEA is concrete and mixed. GPT4.1 was run with no explicit output cap, so the OpenAI default budget of 32,768 completion tokens applied. In 3 of the 9 SEA *find-config* runs the model reached exactly this budget and terminated mid-trace in a repeated reasoning loop, leaving an empty `<answer>` block, a generation-

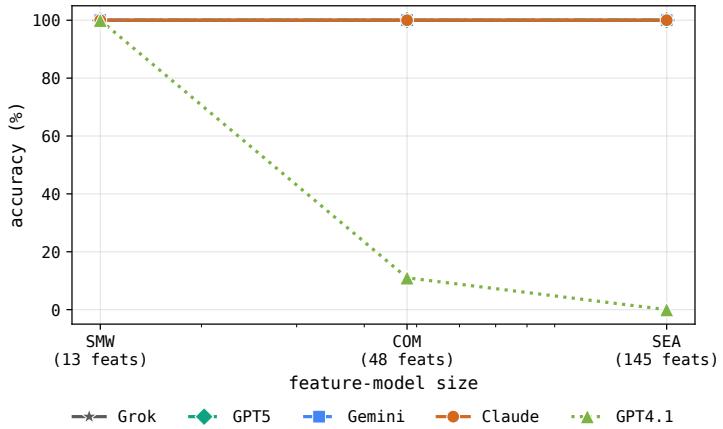


Figure 4: E4b *find-config* accuracy on the satisfiable variants vs. FM size. GPT4.1 collapses from 100% on SMW to 0% on SEA. The four other LLMs remain at ceiling.

budget truncation. In the other 6 runs it returned a complete 145-feature assignment well within budget (1,941 to 13,510 output tokens) that violates at least one constraint, a genuine constraint-reasoning error. The collapse therefore combines budget exhaustion (3 of 9) with constraint-reasoning failure (6 of 9). Budget exhaustion also appears on COM *find-config*, in 4 of 9 runs. We treat this scale-cliff as the strongest finding of RQ-B. A human-side counterpart appears in the comprehension study of Rezaei Sepasi et al. [40]. For human readers, the number of features and cross-tree constraints interact to affect accuracy on configuration-validation tasks, and greater visual effort on the model predicts more errors. The LLM effect here is sharper and narrower, a collapse confined to a single model rather than a graded decline across the selection.

7. World-Knowledge Invocation (RQ-C)

RQ-C is addressed by a single experiment, E5, which asks whether LLMs invoke world knowledge encoded in feature names when reasoning about FMs, and whether the invocation helps or hurts. The manipulation has no human counterpart in [17, 18], because the constraint logic is held fixed and only the feature names vary across conditions.

E5 manipulates the naming of features in two FMs of contrasting semantic richness, COM (48 features, Spanish nouns such as `Mesa`, `Portatil`, `Servidor`) and SEA (145 features, English technical compounds such as `InjectionChokeValve`). See Table 3 for the structural metrics. Three naming variants are tested for each FM, namely *abstract* (features renamed to `f1`…`fN`), *domain-aligned* (the original verbatim names), and *domain-scrambled* (a deterministic bijection over the original name set). Listing 2 shows the same COM sub-tree under all three variants. The structural logic is identical and only the vocabulary changes. Three tasks, *find-config*, *sat*, *count*, are run on each (FM, variant) cell.

Listing 2: The same COM sub-tree, in UVL, under the three E5 naming variants. The structure is identical across variants, and only the feature names differ.

```
abstract                  domain-aligned              domain-scrambled
f1                        Taller1Computadores         Lenovo
  mandatory                 mandatory                   mandatory
    f2                        Tipo                        A1TB
      alternative               alternative                 alternative
        f3                        Mesa                        SO
        f4                        Portatil                    P24
        f5                        Servidor                    Servidor
```

sat is at or near ceiling for all LLMs on both FMs and all three naming variants and contributes no signal. The other two tasks both show an abstract naming effect (Figure 5). Aggregated across the

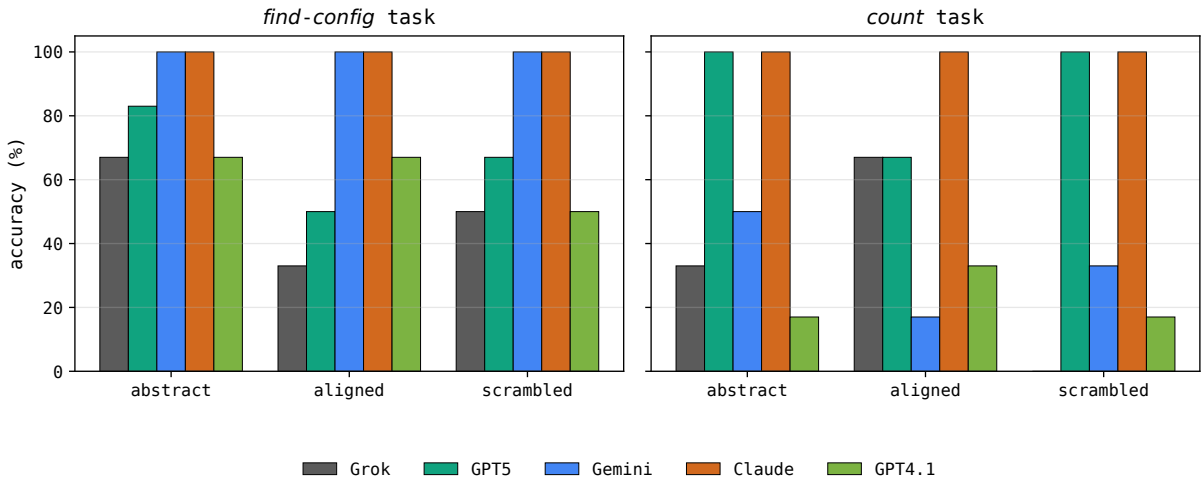


Figure 5: Naming-effect accuracy on E5's *find-config* task (left) and *count* task (right) by naming variant.

Table 11
E5 *find-config* accuracy by FM and naming variant.

FM	Variant	Grok	GPT5	Gemini	Claude	GPT4.1
COM	abstract	1.00	1.00	1.00	1.00	1.00
COM	domain-aligned	0.67	1.00	1.00	1.00	1.00
COM	domain-scramb.	1.00	1.00	1.00	1.00	1.00
SEA	abstract	0.33	0.67	1.00	1.00	0.33
SEA	domain-aligned	0.00	0.00	1.00	1.00	0.33
SEA	domain-scramb.	0.00	0.33	1.00	1.00	0.00

five LLMs on the *find-config* task, accuracy is 0.83 on *abstract*, 0.70 on *domain-aligned*, and 0.73 on *domain-scrambled*. The abstract-vs-aligned gap is 13 pp. On the *count* task, the same aggregation yields 0.60 on *abstract*, 0.57 on *domain-aligned*, and 0.50 on *domain-scrambled*, an abstract-vs-aligned gap of 3 pp. The two LLMs strongest on this task (Claude and Gemini) reach 100% on the *find-config* task across all variants and contribute no signal. The signal is driven by GPT4.1, GPT5, and Grok. The effect is most pronounced on the larger FM (SEA), where Grok and GPT5 drop to 0% on *domain-aligned* from non-zero accuracy on *abstract* (Table 11). The replication across two distinct task types is the central piece of evidence. Interpretation is deferred to Section 8.

Scaffolding ablation. The naming effect is preserved across the four foundational-scaffolding variants of the E5 *find-config* ablation (Figure 6, right panel). The gradient varies across LLMs, but the abstract-vs-aligned gap remains in the same direction across all variants. As a secondary observation, removing the UVL language reference (`no_gram_lang`) shifts the gap from 13 pp to 23 pp. The UVL language reference contains worked examples such as `Windows`, `MobilePhone`, and `Sandwich`, and their removal widens the naming effect. We note this as a supplementary observation.

8. Discussion

8.1. Inverse Profile

Our experiments produce a cognitive-robustness profile that is the inverse of the human one, because the two kinds of subject fail in different places. First, humans fail on constraint form. Logically equivalent rewrites, reorderings, and translations from natural language to a formal encoding increase the cognitive load on the knowledge engineer even though the underlying logic is unchanged, which is

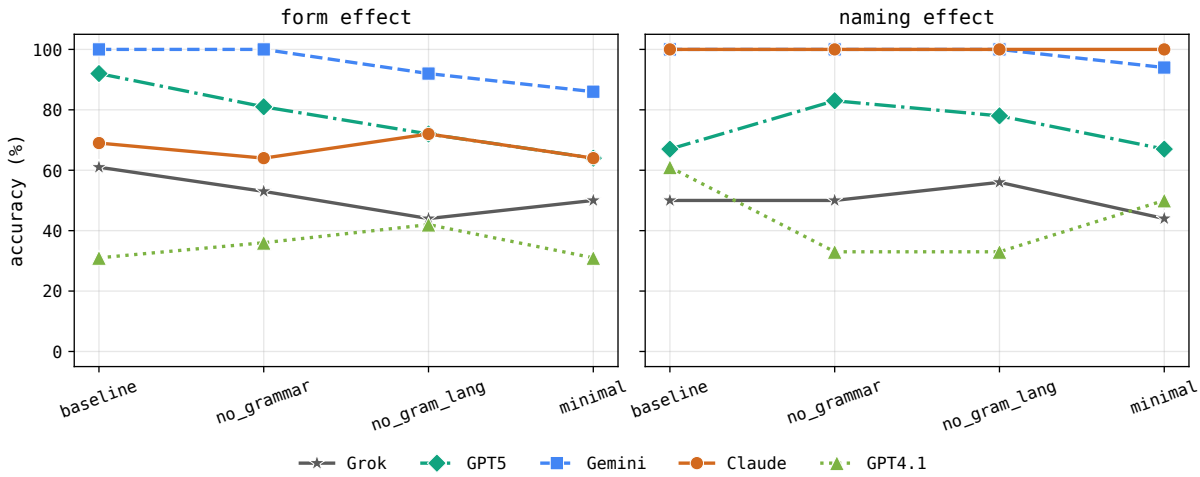


Figure 6: Foundational-scaffolding ablation gradient on E1b (left) and E5 *find-config* (right). GPT5's monotone decline on E1b and GPT4.1's steep drop on E5 `no_grammar` are the two main patterns.

the source of the error patterns reported in [17, 18]. The five LLMs are largely insensitive to these same manipulations, staying at or near ceiling on representation, ordering, and formalization and exceeding the human baseline on redundancy (RQ-A). This is unsurprising in hindsight, because the tested LLMs are pretrained on large volumes of symbolic and formal text, which plausibly makes them robust to the representational variation that induces human errors. Second, LLMs fail on content and capacity. Rather than depending on how a constraint is represented, their failures stem from the world knowledge that meaningful feature names invoke (E5), from natural-language ambiguity that no rewriting removes (the conflation of the *mandatory* relationship in E2b), and from a per-model breakdown at scale that is partly budget exhaustion and partly reasoning error (E4b).

8.2. Capability and Scaffolding Interact

The four-variant foundational-scaffolding ablation shows two distinct interaction patterns (Figure 6). On E1b, among the three reasoning models, GPT5 drops the most when scaffolding is removed (92% at `baseline` to 64% at `minimal`, −28 pp), Gemini drops less (100% to 86%, −14 pp), and Grok is noisy. The two non-reasoning models (Claude, GPT4.1) are also flat or noisy across variants. The reasoning model most dependent on foundational context is therefore GPT5. On E5's *find-config* task, the pattern shifts to the non-reasoning side: GPT4.1 drops steeply when the grammar reference is removed (−28 pp from baseline to `no_grammar`), with the other variants recovering partially. The dependence is on specific scaffolding components rather than on context volume.

8.3. Claude's Combinatorial Reasoning on the Counting Task

Configuration counting on E5 clearly separates the LLM selection (Figure 7). Among the five LLMs, the non-reasoning Claude exact-matches the ground-truth count on 18 of 18 runs across the six (FM, naming variant) cells. The reasoning GPT5 scores 16 of 18. Gemini, GPT4.1, and Grok score 6, 4, and 6 respectively. The strongest combinatorial-counting LLM in this selection is non-reasoning. The SEA configuration space has over 31 billion valid configurations (exactly 31,980,064,896) as computed by FLAMA. The reasoning-trace dump for Claude's run on SEA under the *abstract* variant verifies step-by-step combinatorial arithmetic matching this ground truth, on a stimulus whose feature names are the abstract identifiers `f1`…`f145` rather than the original SubseaCS vocabulary. Gemini's failure mode is structured but mis-counted. The trace records a +396 overcounting at the joint subtree of features `f94` and `f124`, traceable to a specific constraint-propagation gap in its step-by-step procedure. Grok produces both wide overestimates and one catastrophic underestimate ($\log_{10}$ error of −8.8 on the SEA run under the *domain-scrambled* variant, corresponding to an answer 8 orders of magnitude below

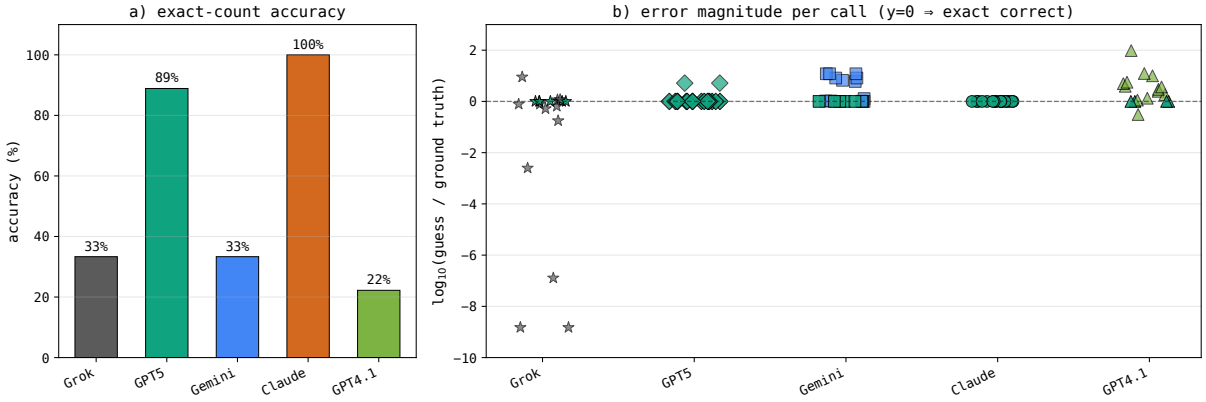


Figure 7: E5 counting-task accuracy (left) and the error magnitude of each call (right). Claude at 18/18 exact. Grok shows one catastrophic underestimate on SEA-scrambled.

ground truth).

8.4. Observations for CKE Tooling

Three observations follow from the results across the LLM selection we tested. First, the LLMs we tested handled abstract identifiers (`f1`…`fN`) better than the original-domain feature names of the same FMs, across two distinct tasks (Section 7, Listing 2). Second, foundational-knowledge scaffolding produced larger accuracy changes on the weaker LLMs in our selection than on the stronger ones (Section 8.2). Third, a capability gap appeared around 100 features on E4b, where one LLM dropped from 100% to 0% across an FM-size step from 13 to 145 features (Section 6.4). These observations describe the behaviour of the five LLMs we tested. Generalisation to other LLMs or to other CKE tasks is not established by this study.

These observations translate into guidance for LLM-assisted authoring pipelines, which we state as proposals rather than validated rules. The naming effect indicates that CKE tooling should neutralise feature vocabulary before an analysis call and re-project the result onto the original names afterwards. The scale-cliff at one LLM indicates that model selection, validated at the target FM size, matters more than reasoning-effort settings when large knowledge bases are at stake. The mandatory-identification failure observed on E2b indicates that blueprint authoring should phrase a mandatory child so that it cannot be read as a cross-tree dependency or a group.

9. Threats to Validity

Sample size. Because LLM outputs are non-deterministic and may differ across runs even under fixed settings, each (experiment, condition, LLM) cell is repeated $n = 3$ times. At this small sample size, the Wilson 95% confidence interval at $\hat{p} = 2/3$ still spans $[0.21,\ 0.94]$, so differences between individual cells are not interpretable as significant. The paper consistently aggregates across stimuli within a condition (typically 15–36 observations per (condition, LLM) cell) and reports rank-order and monotonic-gradient patterns.

Scaffolding interaction with the representation effect. The representation effect at SMW baseline does not survive the removal of foundational scaffolding (Section 6.1). The naming-effect direction is preserved across all four scaffolding variants. We report scaffolding as a moderator of the representation effect but not of the naming effect.

Size and naming are confounded in E1b. SMW is both the smallest FM of E1b and the only one carrying domain-meaningful feature names (Section 4). Since E5 shows that such names shift

accuracy by 3 to 13 pp (Section 7), the SMW-only representation effect of Section 6.1 cannot be attributed to FM size alone, and the present design cannot separate the two axes. A factorial design crossing representation with naming would be needed to do so.

Data contamination. The FMs of E1b, E3b, and E4b are the abstract variants distributed with [27], with the exception of SMW (Section 4), insulating those experiments from contamination by the original-name FMs. Because the E1b representation effect appears on SMW alone, contamination cannot be ruled out by construction for that result. E5 uses original-name FMs by design, because naming is the experimental variable. E2a runs a verbatim-vs-paraphrased test on four tasks and finds no contamination effect.

LLM-selection coverage. Five LLMs from four vendors are tested (Table 4). Three are reasoning-optimized models (Gemini, GPT5, Grok), of which two have explicit reasoning-effort settings (GPT5 at high, Grok at low). Two are non-reasoning models (Claude, GPT4.1). Coverage is not exhaustive. Patterns reported here (the scale-cliff in particular) are tied to specific LLM identities. They should not be read as properties of LLMs in general.

10. Conclusion and Future Work

This paper asked whether the cognitive error patterns that constrain human knowledge engineers also constrain the LLMs now enlisted as analysis assistants, and whether they survive the step from synthetic CSPs to industrial-scale feature models. One half of the answer motivates LLM-assisted authoring. The representation, ordering, and formalization manipulations that derail humans leave all five LLMs at or near ceiling, with a single formalization task as the only exception, and on redundancy detection several clear a baseline that no human participant ever reached. The other half is a caution, because what destabilises LLMs lies elsewhere. One model collapses across an FM-size step that its peers pass without loss of accuracy, domain-meaningful feature names depress accuracy by 3 to 13 pp relative to abstract identifiers (`f1`…`fN`), and the *mandatory* relationship is systematically misidentified, which locates the origin of the semantic-slip failure mode reported by [27]. Each instability calls for a concrete mitigation in the authoring pipeline (Section 8.4). The broader lesson is methodological. The cognitive robustness of an LLM cannot be read off human studies, feasibility results, or confusion signals inferred from a different analysis task, but has its own failure profile that must be characterised directly. Characterising that cognitive-robustness profile, and the points at which it diverges from the human one, is a prerequisite for keeping LLM-assisted knowledge bases trustworthy and comprehensible.

Open topics for future research are the following. First, the representation effect at SMW does not survive the foundational-scaffolding ablation on E1b, which indicates that scaffolding interacts with it in ways the present design cannot decompose. A factorial ablation that crosses the representation effect with each scaffolding component separately is needed. Second, the redundancy detection failure modes identified on E3b (over-flag on Gemini and GPT5, under-flag on GPT4.1) suggest that the underlying causes differ across LLM identities. A causal experiment that varies the size of the no-redundancy control set independently of the true-redundancy set would separate calibration from coverage. Third, the naming-effect finding rests on two FMs and three task types. A larger sweep across the FM dataset of [27], cross-cut with a semantic-similarity metric for each feature between the *domain-aligned* and *domain-scrambled* variants, would isolate which features carry the world-knowledge interference.

Acknowledgments

The work presented in this paper has been developed within the research project GenRE funded by the Austrian Research Promotion Agency under the project number 915086.

Declaration on Generative AI

During the preparation of this work, the authors used Claude in order to check grammar and spelling and to paraphrase and reword text. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] J. McDermott, R1: A rule-based configurer of computer systems, Artificial Intelligence 19 (1982) 39–88. doi:10.1016/0004-3702(82)90021-2.

[2] M. Stumptner, G. Friedrich, A. Haselböck, Generative constraint-based configuration of large technical systems, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12 (1998) 307–320. doi:10.1017/S0890060498124083.

[3] S. Apel, D. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation, Springer, 2013. doi:10.1007/978-3-642-37521-7.

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute, 1990.

[5] R. Reiter, A theory of diagnosis from first principles, Artificial Intelligence 32 (1987) 57–95. doi:10.1016/0004-3702(87)90062-2.

[6] U. Junker, QuickXPlain: Preferred explanations and relaxations for over-constrained problems, in: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004), AAAI Press, 2004, pp. 167–172.

[7] A. Felfernig, M. Schubert, C. Zehentner, An efficient diagnosis algorithm for inconsistent constraint sets, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 26 (2012) 53–62. URL: https://doi.org/10.1017/S0890060411000011. doi:10.1017/S0890060411000011.

[8] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Consistency-based diagnosis of configuration knowledge bases, Artificial Intelligence 152 (2004) 213–234. doi:10.1016/S0004-3702(03)00117-6.

[9] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, M. Toro, Automated error analysis for the agilization of feature modeling, Journal of Systems and Software 81 (2008) 883–896.

[10] V.-M. Le, A. Felfernig, M. Uta, D. Benavides, J. Galindo, T. Tran, DirectDebug: Automated testing and debugging of feature models, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), 2021, pp. 81–85.

[11] M. Sabin, E. C. Freuder, Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems, in: AAAI Workshop on Configuration, Orlando, FL, USA, 1999, pp. 90–94.

[12] A. Felfernig, C. Zehentner, P. Blazek, CoreDiag: Eliminating redundancy in constraint sets, in: Proceedings of the 22nd International Workshop on Principles of Diagnosis (DX 2011), Murnau, Germany, 2011, pp. 219–224.

[13] D. Benavides, S. Segura, A. Ruiz-Cortes, Automated analysis of feature models 20 years later: A literature review, Inf. Sys. 35 (2010) 615–636.

[14] J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, A. Ruiz-Cortés, Automated analysis of feature models: Quo Vadis?, in: Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC '19), ACM, Paris, France, 2019, p. 302. doi:10.1145/3336294.3342373.

[15] A. Felfernig, M. Mandl, A. Pum, M. Schubert, Empirical knowledge engineering: Cognitive aspects in the development of constraint-based recommenders, in: Proceedings of the 23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE 2010), Cordoba, Spain, 2010, pp. 631–640.

[16] S. Shafiee, K. Kristjansdottir, L. Hvam, A. Felfernig, A. Myrodia, Analysis of visual representation

techniques for product configuration systems in industrial companies, in: Proceedings of the 2016 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), IEEE, 2016, pp. 999–1004.

[17] A. Felfernig, S. Reiterer, M. Stettinger, F. Reinfrank, M. Jeran, G. Ninaus, Recommender systems for configuration knowledge engineering, in: M. Aldanondo, A. A. Falkner (Eds.), Proceedings of the 15th International Configuration Workshop, Vienna, Austria, August 29-30, 2013, CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 51–54. URL: https://ceur-ws.org/Vol-1128/paper7.pdf.

[18] A. Felfernig, S. Reiterer, M. Stettinger, J. Tiihonen, Towards understanding cognitive aspects of configuration knowledge formalization, in: K. Schmid, Ø. Haugen, J. Müller (Eds.), Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '15, Hildesheim, Germany, January 21-23, 2015, ACM, 2015, p. 117. URL: https://doi.org/10.1145/2701319.2701327. doi:10.1145/2701319.2701327.

[19] F. Rossi, P. van Beek, T. Walsh, Handbook of Constraint Programming, Elsevier Science, 2006.

[20] D. M. Berry, E. Kamsties, M. M. Krieger, From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity, Technical Report, University of Waterloo, 2003.

[21] 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, in: Advances in Neural Information Processing Systems, volume 33, 2020, pp. 1877–1901.

[22] 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: Advances in Neural Information Processing Systems, volume 35, 2022, pp. 24824–24837.

[23] L. Hotz, C. Bähnisch, S. Lubos, A. Felfernig, J. Twiefel, Exploiting large language models for the automated generation of constraint satisfaction problems, in: Proceedings of the 26th International Workshop on Configuration (ConfWS 2024), volume 3812 of *CEUR Workshop Proceedings*, 2024, pp. 91–100.

[24] A. Ishay, Z. Yang, J. Lee, Leveraging large language models to generate answer set programs, in: Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning (KR), 2023, pp. 374–383. doi:10.24963/kr.2023/37.

[25] J. A. Galindo, A. J. Domínguez, J. White, D. Benavides, Large language models to generate meaningful feature model instances, in: Proceedings of the 27th ACM International Systems and Software Product Line Conference – Volume A (SPLC '23), ACM, Tokyo, Japan, 2023, pp. 15–26. doi:10.1145/3579027.3608973.

[26] M. Parmar, N. Patel, N. Varshney, M. Nakamura, M. Luo, S. Mashetty, A. Mitra, C. Baral, LogicBench: Towards systematic evaluation of logical reasoning ability of large language models, in: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), 2024, pp. 13679–13707. doi:10.18653/v1/2024.acl-long.739.

[27] V.-M. Le, T. N. T. Tran, S. Lubos, A. Felfernig, D. Garber, Early-stage product line validation using LLMs: A study on semi-formal blueprint analysis, in: Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing, SAC 2026, Thessaloniki, Greece, March 23-27, 2026, ACM, 2026, pp. 1620–1627. doi:10.1145/3748522.3779903.

[28] J. A. Galindo, J.-M. Horcas, A. Felfernig, D. Fernández-Amorós, D. Benavides, FLAMA: A collaborative effort to build a new framework for the automated analysis of feature models, in: Proceedings of the 27th ACM International Systems and Software Product Line Conference – Volume B (SPLC '23), ACM, Tokyo, Japan, 2023, pp. 16–19. doi:10.1145/3579028.3609008.

[29] D. Benavides, C. Sundermann, K. Feichtinger, J. A. Galindo, R. Rabiser, T. Thüm, UVL: Feature modelling with the universal variability language, Journal of Systems and Software 225 (2025) 112326. doi:10.1016/j.jss.2024.112326.

[30] M. H. Liffiton, K. A. Sakallah, Algorithms for computing minimal unsatisfiable subsets of constraints, Journal of Automated Reasoning 40 (2008) 1–33.

[31] V.-M. Le, A. Felfernig, M. Uta, T. N. T. Tran, C. Vidal Silva, WipeOutR: automated redundancy detection for feature models, in: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A, SPLC '22, Association for Computing Machinery, New York,

NY, USA, 2022, p. 164–169.
[32] A. Felfernig, V.-M. Le, A. Popescu, M. Uta, T. N. T. Tran, M. Atas, An overview of recommender systems and machine learning in feature modeling and configuration, in: Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS), 2021.
[33] A. Popescu, S. Polat-Erdeniz, A. Felfernig, M. Uta, M. Atas, V.-M. Le, K. Pilsl, M. Enzelsberger, T. N. T. Tran, An overview of machine learning techniques in constraint solving, Journal of Intelligent Information Systems 58 (2022) 91–118.
[34] K. Michailidis, D. Tsouros, T. Guns, Constraint modelling with LLMs using in-context learning, in: 30th International Conference on Principles and Practice of Constraint Programming (CP 2024), volume 307 of *LIPIcs*, Schloss Dagstuhl, 2024, pp. 20:1–20:27. doi:10.4230/LIPIcs.CP.2024.20.
[35] L. Marchezan, E. Rodrigues, W. K. G. Assunção, M. Bernardino, F. P. Basso, J. Carbonell, Software product line scoping: A systematic literature review, in: Proceedings of the 26th ACM International Systems and Software Product Line Conference – Volume A (SPLC '22), ACM, Graz, Austria, 2022, p. 256. doi:10.1145/3546932.3547012.
[36] xAI, Models and pricing, 2025. URL: https://docs.x.ai/docs/models, accessed: 2025-09-29.
[37] OpenAI, OpenAI models, 2025. URL: https://platform.openai.com/docs/models, accessed: 2025-09-29.
[38] Google, Gemini models, 2025. URL: https://ai.google.dev/gemini-api/docs/models, accessed: 2025-09-29.
[39] Anthropic, All models overview, 2025. URL: https://docs.claude.com/en/docs/about-claude/models/overview, accessed: 2025-09-29.
[40] E. Rezaei Sepasi, K. Nezami Balouchi, J. Mercier, R. E. Lopez-Herrejon, Towards a cognitive model of feature model comprehension: An exploratory study using eye-tracking, in: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A (SPLC '22), ACM, 2022. doi:10.1145/3546932.3546995.

A. Counting-Level Cross-Check for E2b

As a counting-level cross-check on the E2b result (Section 6.2), we analyse the count-error data of [27]. For each (LLM, FM) cell, we compute the miscounts of the six relationship and constraint types (*mandatory*, *optional*, *or*, *alternative*, *requires*, *excludes*) against FLAMA ground truth. We then compute Pearson correlations between the miscount vectors of each pair of relationship and constraint types, on two slices, i.e., all 10 FMs ($n = 112$ cells) and the 8 medium-scale FMs only ($n = 96$ cells, which exclude the two industrial-scale FMs CNN1 and CNNf whose token counts saturate the prompts of [27] and dominate the Pearson computation as scale outliers). Table 12 reports the filtered-slice matrix.

Table 12
Pearson correlations between miscount vectors of the six FM relationship and constraint types in the count-error data of [27], filtered slice ($n = 96$, excluding CNN1 and CNNf). A negative entry is consistent with the LLM confusing the two types, i.e., over-counting one goes with under-counting the other. The two strongest negative pairs (bold) are *or*–*alternative* and *mandatory*–*or*.

	mand.	*opt.*	*or*	*alt.*	*req.*	*excl.*
mandatory	1.000					
optional	0.026	1.000				
or	**−0.364**	0.013	1.000			
alternative	0.200	0.011	**−0.578**	1.000		
requires	0.020	−0.090	−0.024	0.003	1.000	
excludes	−0.023	−0.026	−0.011	−0.026	0.024	1.000

On the filtered slice, the strongest negative correlation holds between *or* and *alternative* miscounts

($r = -0.578$), followed by *mandatory* and *or* ($r = -0.364$). A negative correlation between two miscount vectors is consistent with the LLM confusing the two types, i.e., over-counting one goes with under-counting the other. These counting-level signals predict the *or–alternative* and *mandatory–or* pairs as the most likely confusion sites at the identification level.

The identification results in Section 6.2 do not match this counting-level prediction. The *or–alternative* pair, i.e., the strongest counting correlation, is identified cleanly (the group class at ceiling). The mandatory fragment does fail, as the *mandatory–or* correlation hints, but the dominant confusion there is with cross-tree *requires*, a pair the correlation leaves near zero. We read this mismatch as evidence that cross-task inferences from counting data are not reliable, and that identification-level testing is required to locate a blueprint-vocabulary redesign target.

The signed net miscount tells a different counting-level story again. *Mandatory* features are net under-counted and reappear mostly as *optional*, but at the identification level the *optional* reading of "must have" is the one the LLMs reject. Neither the symmetric correlation nor the signed net flow therefore anticipates the identification result, i.e., the conflation of *mandatory* with cross-tree *requires*.

Counting and identification are not independent. To count a relationship or constraint type the LLM must identify each instance and then enumerate and tally the instances across the whole feature model, so a miscount can arise above the identification layer, in enumeration, in arithmetic, or in the scale of an industrial FM. A counting-level confusion signal therefore need not reflect an identification confusion. The *or*-versus-*alternative* pair makes this concrete, since its strong negative miscount correlation coexists with flawless identification (the group class at ceiling), which places that miscount in the enumeration and tallying step rather than in the reading of the relationship. Because the counting signals come from the twelve-LLM evaluation of [27] while our identification runs use a different five-LLM set, the observed non-transfer also reflects differences between models rather than a controlled within-model result.

Authoring Automotive Configurations with Internal DSLs

Carsten Sinz

Karlsruhe University of Applied Sciences, Moltkestraße 30, 76133 Karlsruhe, Germany

Abstract

Configuration knowledge in the automotive domain must integrate a growing range of engineering and market concerns, and is increasingly maintained, reviewed, tested, and deployed like software. It combines classical product variability with market regulations, homologation rules, packages and trims, software-enabled functions, manufacturing constraints, supply-chain restrictions, and post-sale activation logic. This position paper argues that internal domain-specific languages (internal DSLs), i.e., domain-oriented APIs embedded in modern host languages such as Swift or Kotlin, are a promising authoring layer for such configuration spaces. They provide typed abstractions, reuse mechanisms, IDE support, and integration into software engineering workflows. However, the Turing completeness of host languages can undermine the declarative semantics needed for solver integration, reproducible analysis, and explanation. We therefore propose a layered approach in which internal-DSL declarations are compiled into a declarative intermediate representation that can be validated, analyzed, exported, and translated to reasoning engines and downstream engineering backends, including testing, documentation, bill-of-materials generation, and manufacturing checks. This is an architectural position supported by illustrative language sketches, not a complete IR specification, solver implementation, or industrial evaluation.

Keywords

Configuration, Variability modeling, Domain-specific languages, Automotive configuration, Feature models

1. Introduction

Product configuration has long been studied as a knowledge-based design task: a configurable product is composed from predefined component types, and admissible combinations are determined by constraints. This view remains central. However, in industrial automotive settings the artifact to be maintained is no longer only a feature hierarchy with cross-tree constraints. It is a heterogeneous body of configuration knowledge that connects hardware variants, software features, regional regulations, homologation rules, sales packages, trims, supply-chain availability, manufacturing constraints, post-sale activation, and over-the-air updates [1].

In such settings, configuration knowledge behaves increasingly like a software artifact. It has versions, owners, dependencies, generated fragments, regression tests, release processes, and downstream consumers. Changes are often small but safety-critical or commercially relevant: a market restriction may invalidate an option package; a supply issue may temporarily remove a component; or a software feature may become available only after a regulatory approval. Maintaining this knowledge with spreadsheets, ad-hoc rule formats, or disconnected modeling tools creates a gap between configuration engineering and the software engineering infrastructure already used in the organization. [2, 3]

In this paper we argue that internal domain-specific languages (DSLs) embedded in modern host languages are a promising authoring layer for automotive configuration spaces. Languages such as Swift and Kotlin support DSL-like syntax through result builders or type-safe builders, while retaining type checking, modularity, IDE support, package management, and test frameworks. These facilities can make configuration models easier to construct, review, and evolve.

The goal is not to move configuration semantics into arbitrary program code. Instead, the host-language DSL should serve as a front end whose declarations are compiled into an explicit, declarative intermediate representation (IR). The IR is the semantic contract: it can be validated, translated to

ConfWS'26: 28th International Workshop on Configuration, Aug 15–16, 2026, Bremen, Germany
carsten.sinz@h-ka.de (C. Sinz)
0000-0001-9718-1802 (C. Sinz)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

solvers, exported to interchange formats, compared across versions, and used for explanations. The host language provides authoring power; the IR preserves analyzability.

The contribution claimed is therefore architectural rather than a new DSL embedding mechanism: it combines automotive-specific configuration concepts and host-language integration with an explicit semantic boundary for analysis, interchange, and traceability.

The proposal is assessed against three requirements: R1, the authoring layer must express automotive concepts beyond a Boolean variability core; R2, it must support abstraction, integration, testing, and modular evolution in software-engineering workflows; and R3, its meaning must remain explicit, analyzable, and portable. The paper develops a Swift-style internal DSL and declarative IR as a position toward these requirements and identifies the research challenges that remain.

2. Background and Related Work

A *configuration model* describes a space of admissible products or systems. It contains selectable entities, attributes, structural relationships, and constraints. A *concrete configuration* is an assignment of decisions and attribute values for one product instance. It may be *complete*, if all required decisions have been made, or *partial*, if some decisions are still open. Validation asks whether a model is well-formed, whether a concrete or partial configuration is consistent with the model, and whether additional decisions can complete it. Interactive configuration, repair, optimization, and diagnosis build on these basic reasoning tasks [4].

Feature modeling is one of the most influential foundations for variability modeling. The FODA report introduced feature-oriented domain analysis as a way to capture commonality and variability in product families [5]. Automated analysis of feature models has since become a mature area, including satisfiability, dead-feature detection, valid product counting, and explanation [6]. FeatureIDE provides an example of an extensible tool environment for feature-oriented software development [7].

The Universal Variability Language (UVL) aims to provide a common textual language for feature models and variability artifacts [8]. It is part of a broader discussion on textual variability modeling languages [9] that aims at making variability models portable and declarative. Clafer [10] is another textual language that combines feature modeling with class-like structures, references, constraints, and solver-backed reasoning. The approach proposed here is complementary: host-language DSLs serve as the authoring layer, while the declarative IR remains the semantic artifact. Solver-oriented modeling languages, such as MiniZinc, demonstrate a complementary tradition: a modeling language can provide a disciplined surface syntax for translation to multiple solving technologies [11, 12]. Automotive hardware/software co-configuration also shows the need for richer solver-facing representations beyond propositional feature constraints [13]. These lines of work motivate a crucial requirement for the approach proposed here: even if a configuration model is authored through a host-language DSL, its meaning should ultimately be represented in a declarative form.

Internal or embedded DSLs are an established language-engineering technique [14]; Scala has likewise been used to embed formal modeling languages [15]. Modern programming languages have made this technique more practical. An internal DSL is embedded in a *host language*, i.e., a general-purpose programming language that provides the DSL's syntax, type system, module system, and execution environment. We use Swift and its result builders [16] in our examples; Kotlin offers a comparable mechanism through type-safe builders [17].

Our position is not that internal DSLs are novel or that configuration should become "just programming". The specific proposal is to use established embedding techniques for typed authoring, generation, modularity, and integration of heterogeneous automotive knowledge, while making a separate declarative IR the analyzable and portable semantic contract.

Table 1 summarizes complementary strengths of the main approaches; "++" denotes strong built-in support, "+" useful support, "○" support through extensions, and "–" a relative weakness.

Table 1
Comparison by requirements relevant to this proposal.

Approach	Explicit semantics	IDE and testing	Abstraction and generation	Portability	Automotive extensibility
UVL	++	+	−	++	○
FeatureIDE	++	++	+	+	○
MiniZinc	++	+	+	+	○
Internal DSL + IR	+	++	++	+	++

3. Internal DSLs as Authoring Languages

Internal DSLs are libraries whose API and syntax are shaped so that client code reads like a domain language. In contrast to external DSLs, they do not require a separate parser, editor, type checker, or build tool. They reuse the host-language ecosystem. For configuration engineering, this reuse can be substantial: model fragments can be packaged, generated, reviewed in code review systems, checked by continuous integration, and tested with ordinary unit tests.

This reuse concerns authoring only; IR validators, solver translations, semantic comparison, and explanation tooling remain configuration-specific infrastructure.

Listings 1, 3, 4, and 5 form one running example: model, generated IR, reusable rules, and vehicle order. Listing 1 shows a small Swift-embedded DSL for an automotive configuration space. The example is intentionally minimal, but it illustrates several properties that are useful in larger models: hierarchical structure, typed quantities, named constraints, market conditions, and rule declarations. The syntax is Swift code, yet the expressions construct data structures rather than directly performing reasoning.

Listing 1: Sketch of a Swift-embedded automotive configuration DSL.

```
let evSUV = Vehicle("EV-SUV") {
    // Feature groups define the configuration vocabulary.
    Feature("Battery", selection: .exactlyOne) {
        Option("Standard", capacity: 75.kWh)
        Option("LongRange", capacity: 100.kWh)
    }

    Feature("Drive", selection: .exactlyOne) {
        Option("RWD")
        Option("AWD")
    }

    // Markets are modeled explicitly, not hidden in generator logic.
    Feature("Market", selection: .exactlyOne) {
        Option("EU")
        Option("US")
        Option("China")
    }
    Feature("WinterPackage", selection: .optional)

    // Rules remain declarative although they are authored in Swift.
    Requires(.option("AWD"), .option("LongRange"))
    Requires(.market("EU"), .feature("eCall"))

    // Typed units make accidental dimension errors less likely.
    Constraint("weight-limit") {
        Attribute.totalMass <= 2_800.kg
    }
}
```

Here the `selection` argument states whether a feature group selects exactly one child or is itself optional. The listing sketches the intended API. The accompanying proof of concept currently constructs

hierarchy, attributed options, binary requirements, and simple numeric constraints; explicit group cardinalities, complete IR serialization, and solver translation remain proposed extensions.

UVL represents this variability core more concisely and with explicit declarative semantics. The internal DSL instead contributes typed generation, reuse, testing, and API integration; the IR is needed to recover a portable semantic artifact.

Internal DSLs are particularly useful when parts of the model are structured or repetitive. A family of option packages can be constructed from tabular product data. A reusable rule template can express that a market-specific regulation requires a certain safety function. Constants can define shared thresholds such as mass limits, battery capacities, or price classes. Such mechanisms are common in software engineering but often awkward in standalone configuration languages. Listings 4 and 5 below return to these examples for reusable rule templates and concrete configurations.

Listing 2: UVL projection of the battery feature.

```
features
  EV_SUV
    mandatory
      Battery
        alternative
          Standard
          LongRange
constraints
  AWD => LongRange
```

4. Layered Architecture and Intermediate Representation

Figure 1 sketches the intended architecture. The DSL is an authoring layer whose declarations are compiled into a declarative IR for analysis and integration. Analysis and integration tools consume this IR rather than depending on how the host-language program happened to execute.

Listing 3: Generation-time loop with explicit generated IR.

```
let chargingStandards = [
    ("EU", "CCS2"),
    ("US", "NACS")
]

let charging = FeatureGroup("Charging") {
    for (market, standard) in chargingStandards {
        Feature("Charging-\(market)") {
            AppliesTo(.market(market))
            Option(standard)
        }
    }
}

// compile() closes the generation-time phase and produces IR.
let ir = charging.compile()

// emit() serializes the IR; the comments show its textual form.
ir.emit()
// feature Charging-EU { appliesTo market("EU"); option CCS2 }
// feature Charging-US { appliesTo market("US"); option NACS }

// Unsafe: model meaning depends on mutable external state.
let liveRules = RegulatoryAPI.currentRules()
if Date.now >= launchDate { enable("NewPackage") }
```

This layering is the central design principle. The DSL may use host-language constructs to assemble the model, but the resulting object must belong to a restricted semantic vocabulary. A loop that generates one feature per market, as in Listing 3, is harmless if the generated IR is explicit. A function that hides an unbounded search or reads mutable external state is problematic if its effect cannot be reproduced or explained. The IR is not a restricted subset of the host-language syntax, but a separate declarative data model produced by the DSL pipeline and used as the semantic artifact for validation, translation, explanation, and downstream engineering tools.

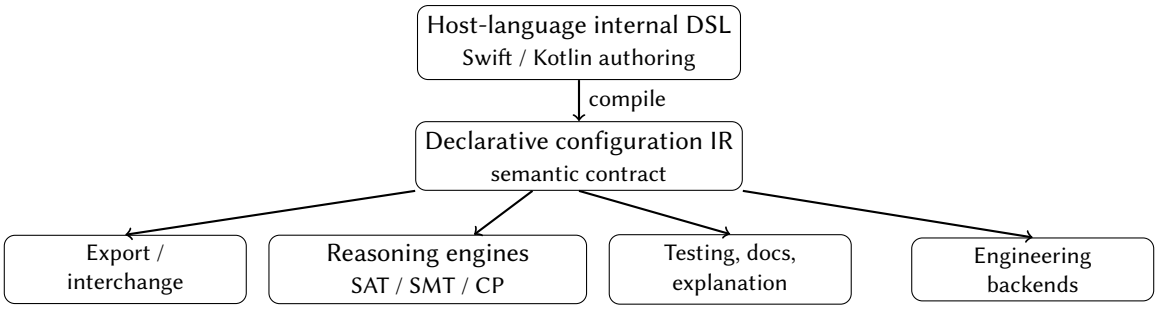


Figure 1: Layered architecture: internal DSL authoring with a declarative IR as semantic artifact for reasoning and engineering backends.

In Listing 3, `compile()` denotes the DSL pipeline step that turns the Swift authoring objects into this separate IR. `emit()` denotes a serializer or pretty-printer for the IR. The IR is shown as comments in the listing only to keep the generated artifact next to the authoring code; in an implementation it would typically be written as JSON, protobuf, or another structured exchange format.

The loop is safe because explicit input is materialized in the IR; live service responses or the system clock can change model meaning without a source revision.

An automotive configuration IR should include at least the following elements: entities such as features, options, components, packages, and software functions; hierarchical and compositional relationships; attributes with typed domains and units; constraints over selections and attributes; named rule groups; provenance links to source code, product data, requirements, or regulations; and annotations for ownership, validity periods, markets, and lifecycle state. For solver integration, the IR should have a clear logical interpretation and a finite, explicit structure that does not depend on re-executing arbitrary host-language code. For engineering integration, it should be serializable, diff-friendly, and stable across tool versions, while preserving enough source-level information to support diagnostics and explanations.

The main benefits of this approach are pragmatic. First, it integrates configuration models with industrial software workflows: version control, review, testing, packaging, and continuous integration. Second, it provides static checks that ad-hoc rule formats often lack; for example, typed quantities can prevent mixing mass, price, and energy values. Third, it offers good support for hierarchical and compositional product structure. Fourth, it reuses existing IDE features such as completion, navigation, and refactoring. Fifth, it can bridge configuration models and implementation artifacts, for example when the same identifiers are used by configurators, software feature flags, and test infrastructure. Sixth, it allows automotive-specific concepts to be added as library abstractions without changing a standalone language grammar.

A passing CI build additionally requires a well-formed, satisfiable IR, successful regression configurations, and an explicit semantic difference from the previous baseline.

The approach also has clear risks. Unrestricted host-language computation can make the model difficult to understand, reproduce, or translate to a solver. The formal semantics of an internal DSL may be less obvious than that of a standalone language. Domain experts may not want to edit code. Tooling can couple configuration knowledge to a vendor ecosystem. Portability may suffer if the IR is not treated as the primary semantic artifact. These risks do not refute the approach; they define its design constraints.

5. Automotive Concerns Beyond Feature Models

Generic variability constructs are necessary but not sufficient for the automotive domain. A vehicle configuration is tied to many concerns that are not naturally captured by a pure feature tree. A configuration model may need to express that a software function is available only in specific markets, that a component is valid only after a production date, that a package implies a homologation-relevant

safety system, or that a factory cannot build a combination because of workstation tooling [2, 3].

Several categories appear repeatedly in practice. The first is *attributes with units and derived values*: mass, energy capacity, range, emissions class, towing capacity, price, and electrical load. The second is *market and regulatory context*: regional availability, homologation, mandatory equipment, language packages, charging standards, and data-protection constraints. The third is *commercial structure*: trims, packages, campaigns, sales channels, and guided selling rules. The fourth is *production and supply*: bill of materials generation, factory feasibility, workstation requirements, tooling, procurement risk, and temporary availability. The fifth is *software lifecycle*: feature activation, subscriptions, OTA updates, compatibility with hardware revisions, and post-sale reconfiguration.

These concerns suggest that configuration models should not be isolated from other artifacts. Traceability is essential. A rule may originate in a regulation, an engineering requirement, a supplier notice, or a sales decision. If a configuration is rejected, the explanation should be able to point back to the responsible rule and its source. If a model changes, impact analysis should identify affected packages, markets, tests, documentation, production processes, and already sold configurations.

Host-language DSLs can help with the repetitive and structured parts of this work. They can avoid repeated declarations by generating similar fragments from data. They can define reusable rule templates, such as “market requires feature” or “package requires option”. They can use constants and formulas for derived attributes. They can construct conditional model fragments for product lines, model years, or markets. They can express concrete configurations from option codes or order data. They can also support documentation generation and regression tests close to the model.

Listing 4 shows a reusable rule template for recurring market constraints. The template function is an ordinary host-language function, but its result is still a declarative DSL fragment. Its benefit is not only shorter syntax: each invocation creates a uniform rule group with the constraint, source metadata, and diagnostic text; related helpers can generate matching regression-test fragments. Similar templates can capture package or campaign rules and make repeated configuration knowledge more consistent and easier to audit.

Listing 4: Reusable rule templates with generated metadata.

```
func marketRequires(_ market: String, _ feature: String,
                    source: String) -> RuleGroup {
    RuleGroup("regulation-\(source)") {
        Requires(.market(market), .feature(feature))
            .source(source)
            .diagnostic("\(feature) is mandatory in \(market)")
    }
}

let regulatoryRules = RuleGroup("regulatory") {
    marketRequires("EU", "eCall", source: "EU-2018-858")
    marketRequires("US", "TirePressureMonitoring",
                   source: "FMVSS-138")
}

let temporalRules = RuleGroup("model-year-rules") {
    Available(.option("LongRange"))
        .valid(from: .modelYear(2027),
               through: .modelYear(2029))
}
```

Temporal validity is evaluated against an explicit model year or production date, never the host computer’s current time.

Concrete vehicle orders can use the same DSL surface, but they represent configuration instances rather than model knowledge. Listing 5 sketches one such order, which can be validated against the generated IR and diagnosed using the reusable rules from Listing 4.

The expressive power of host-language DSLs requires discipline. Programming constructs should build declarative model fragments, not hide semantic decisions in arbitrary computations. A useful

discipline is separating *model generation time* from *model meaning*. At generation time, loops, helper functions, and imports may assemble the model. Once assembled, the IR should contain an explicit set of declarations, constraints, attributes, and provenance links. Reasoning tools should not depend on re-executing arbitrary code in order to understand the model.

Listing 5: Concrete configuration instance.

```
let order4711 = Configuration("order-4711", model: evSUV) {
    Select.market("EU")
    Select.option("LongRange")
    Select.option("AWD")
    Select.option("WinterPackage")
}
```

Listing 3 already illustrates this generation-time discipline and the explicit IR that closes the semantic boundary.

Beyond validation, such an IR can support further configuration tasks: repair and explanation, preference-based optimization, model diffing, semantic equivalence checking, migration of existing configurations, impact analysis, test generation, and documentation generation. It can also act as a bridge to industrial applications such as bill-of-materials generation, manufacturing process planning, production scheduling, simulation setup, certification analysis, quote generation, and service manual generation.

6. Research Challenges

Turning this layered architecture into a dependable modeling practice requires answers to several research questions:

Semantic restriction. The central question is how expressive the authoring DSL can be while preserving analyzability. A useful DSL should allow abstraction, reuse, and generation, but it should also make clear which host-language constructs affect only generation and which constructs have semantic meaning in the model. Static analyses or linting rules may be needed to reject patterns that make the resulting IR non-reproducible or opaque.

IR design. A configuration IR for automotive use must be more than a Boolean feature model, but it must remain simple enough for translation to solvers and interchange formats. Open design questions include how to represent typed attributes, temporal validity, provenance, market scopes, optional software activation, and manufacturing constraints without producing a language that is too broad to reason about.

Scalability. Scalability has at least three dimensions. Model size concerns the number of features, options, attributes, and rules. Lifecycle scale concerns the number of versions, markets, product years, and dependent artifacts. Organizational scale concerns ownership, reviews, approvals, and division of responsibility across departments. An internal DSL may help with the second and third dimensions; solver scalability is often manageable with current reasoning technology, but the generated IR still must preserve the structure needed for efficient translation and incremental analysis. Team-owned, versioned packages can hold battery, market, manufacturing, and software fragments; integration builds compose their IR and check identifiers, dependencies, and constraints.

IR tooling. Host-language IDE and build support does not replace versioned IR schemas, validators, solver backends, semantic comparison, source navigation, and explanation.

Explanation and traceability. Solver-level explanations often refer to low-level constraints, whereas engineers and domain experts need explanations in terms of source-level rules, regulations, packages, and product concepts. The translation from DSL to IR to solver must preserve traceability in both directions.

Testing and regression. If configuration knowledge is software, then it needs tests. Regression suites should check that known valid configurations remain valid, known invalid configurations remain invalid, and intended model changes have the expected impact. Semantic diffing and equivalence checking are especially important when a small source change generates many IR-level changes.

Human authoring. A host-language DSL may be comfortable for software engineers but less comfortable for domain experts. Research is needed on hybrid workflows in which domain experts edit structured data, forms, or controlled tables, while the DSL provides the typed integration layer and the IR provides common semantics.

7. Discussion

The proposal bridges two established ways of modeling configuration knowledge. One approach is to model automotive configuration purely in general-purpose program code. This would provide maximum expressiveness but weak guarantees about semantics, portability, and solver translation. The other approach is to insist that all configuration knowledge must be written directly in a standalone declarative language. This protects analyzability, but it can make abstraction, generation, software workflow integration, and domain-specific extension harder than necessary.

The layered approach combines the useful parts of both positions. The internal DSL is a productive authoring interface. The IR is the formal artifact. Solvers, exporters, documentation tools, and industrial backends operate on the IR. This division also makes it possible to support multiple authoring surfaces over time: Swift or Kotlin libraries for engineering teams, generated fragments from product data, and perhaps graphical or tabular editors for domain experts.

Against the requirements stated in the introduction, the sketch addresses R1 through automotive entities, attributes, units, and scoped rules, although a complete metamodel remains open. It addresses R2 by reusing host-language modules, tests, generation, and integration. R3 is the central qualification: analyzability and portability depend on compiling every declaration into an explicit IR with preserved provenance; formal IR semantics and verified solver translations remain future work.

8. Conclusion

Automotive configuration spaces are increasingly maintained as software artifacts. They require more than a generic variability core: attributes, units, market scopes, regulatory constraints, manufacturing feasibility, software activation, traceability, and lifecycle information all matter. Modern host-language DSLs offer a promising way to author such models because they bring types, modularity, reuse, IDE support, and testing infrastructure to configuration engineering.

The main design requirement is that DSL declarations must be compiled into a declarative intermediate representation rather than making the host-language program itself the semantic artifact. This IR should preserve the logical meaning needed for reasoning and the provenance needed for explanation and maintenance. The resulting research agenda includes IR design, semantic restrictions on DSLs, traceable solver translation, regression testing, model evolution, and hybrid authoring workflows for engineers and domain experts.

The position advanced here is modest but consequential: host-language DSLs should be explored as executable specification front ends for configuration spaces, provided that their expressive power is channeled into explicit, analyzable, and portable configuration models.

Declaration on Generative AI

During the preparation of this work, the author used ChatGPT and OpenAI Codex to support drafting, editing, and Swift code generation. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of this publication.

References

[1] D. Bischoff, C. Sinz, Heterogeneity: A challenge in automotive product configuration, in: Proc. of ConfWS 2025, CEUR Workshop Proc., 2025, pp. 96–102. URL: https://ceur-ws.org/Vol-4149/paper8.pdf.

[2] P. Zellmer, L. Holsten, T. Leich, J. Krüger, Product-structuring concepts for automotive platforms: A systematic mapping study, in: Proc. of SPLC 2023, ACM, 2023, pp. 170–181. doi:10.1145/3579027.3608988.

[3] P. Ochs, T. Pett, I. Schaefer, Consistency is key: Can your product line realise what it models?, in: Proc. of MODELS 2024, ACM, 2024, pp. 690–699. doi:10.1145/3652620.3687812.

[4] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen, Knowledge-Based Configuration: From Research to Business Cases, Morgan Kaufmann, 2014.

[5] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, Software Engineering Institute, Carnegie Mellon University, 1990.

[6] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: A literature review, Information Systems 35 (2010) 615–636. doi:10.1016/j.is.2010.01.001.

[7] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, FeatureIDE: An extensible framework for feature-oriented software development, Science of Computer Programming 79 (2014) 70–85. doi:10.1016/j.scico.2012.06.002.

[8] L. Michael, C. Sundermann, M. Schubanz, S. Greiner, M. Nieke, R. Schröter, W. Fenske, T. Thüm, G. Saake, UVL: Feature modelling with the universal variability language, Journal of Systems and Software 225 (2025) 112326. doi:10.1016/j.jss.2024.112326.

[9] M. H. ter Beek, K. Schmid, H. Eichelberger, Textual variability modeling languages: An overview and considerations, in: Proc. of SPLC 2019, ACM, 2019, pp. 151–157. doi:10.1145/3307630.3342398.

[10] K. Bąk, Z. Diskin, M. Antkiewicz, K. Czarnecki, A. Wąsowski, Clafer: Unifying class and feature modeling, Software and Systems Modeling 15 (2016) 811–845. doi:10.1007/s10270-014-0441-1.

[11] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, MiniZinc: Towards a standard CP modelling language, in: Proc. of CP 2007, Springer, 2007, pp. 529–543. doi:10.1007/978-3-540-74970-7_38.

[12] G. Tack, R. Comploi-Taupe, A. Falkner, G. Schenner, MiniZinc with objects, Constraints 30 (2025) 109–137. doi:10.1007/s10601-025-09380-3.

[13] F. Jost, C. Sinz, Handling automotive hardware/software co-configurations with integer difference logic, in: Proc. of VaMoS 2024, ACM, 2024, pp. 103–111. doi:10.1145/3634713.3634728.

[14] P. Hudak, Building domain-specific embedded languages, ACM Computing Surveys 28 (1996) 196. doi:10.1145/242224.242477.

[15] C. Artho, K. Havelund, R. Kumar, Y. Yamagata, Domain-specific languages with Scala, in: Proc. of ICFEM 2015, volume 9407 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 1–16. doi:10.1007/978-3-319-25423-4_1.

[16] Apple Inc., The Swift programming language: Result builders, https://docs.swift.org/swift-book/documentation/the-swift-programming-language/advancedoperators#Result-Builders, 2026. Accessed 2026-05-27.

[17] JetBrains, Type-safe builders, https://kotlinlang.org/docs/type-safe-builders.html, 2026. Accessed 2026-05-27.