Machine Learning-Based Generalization Queries for Constraint Acquisition

Dimos Tsouros
UOWM, Greece

Senne Berden
KU Leuven, Belgium

Tias Guns
KU Leuven, Belgium

Abstract

Interactive Constraint Acquisition (CA) can assist in modeling constraint problems by acquiring constraints through a query-based interaction with the user. However, existing methods often require a large number of queries. We aim to address this limitation by introducing constraint-specification-based generalization queries, which lift individual learned constraints to higher-level specifications that capture entire sets of constraints at once. We then propose GQ-Gen, a machine learning-based method to extract such specifications from the learned and excluded constraints during the acquisition process and show how it can be integrated directly within any existing interactive CA algorithm. Our experiments show that our proposed approach reduces the number of queries substantially (by up to 95%), while our user study demonstrates that LLM-based natural language formulations of our generalization queries are understandable to users.

2012 ACM Subject Classification Computing methodologies → Discrete space search; Computing methodologies → Supervised learning by classification; Theory of computation → Constraint and logic programming; Computing methodologies → Active learning settings

Keywords and phrases Constraint acquisition, Machine learning, Generalization Queries, Modeling

Acknowledgements The research received funding from the framework of H.F.R.I call "4th Call for H.F.R.I.'s Research Projects to Support Postdoctoral Researchers" (H.F.R.I. Project Number: 28553); from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme; from the Research Foundation-Flanders (FWO) (Grant No. 11PQ024N);

1 Introduction

Constraint Programming (CP) is one of the key paradigms for solving combinatorial problems. In CP, the user first models the problem by declaratively stating constraints over a set of decision variables, after which a solver is used to find a solution. Although this model+solve paradigm has been successful, the modeling process can be challenging and is considered a major bottleneck to the broader adoption of CP [13, 11, 12].

As a result, a considerable body of research focuses on making CP modeling more accessible, including the development of solver-agnostic modeling languages [23, 14, 15], and LLM-based assistants [29, 22, 17]. In addition to these, the topic of *Constraint Acquisition (CA)* has received a lot of attention, aiming to acquire the model of a problem from data. A recent application of CA in a large-scale real-world problem showed its significant potential [1]. In *passive* CA, the system uses a dataset of pre-existing examples of solutions and non-solutions to acquire a set of constraints [2, 8, 18, 3]. In *interactive* acquisition, the system instead interacts with the user [7, 4] to learn a set of constraints that represents the problem the user has in mind. Most state-of-the-art interactive CA algorithms are based on the use of *(partial) membership queries* [6, 19, 31, 33]. These queries ask the user to classify (partial) assignments to the variables as valid or not.

One of the most significant drawbacks of interactive CA is the still substantial number of queries needed, which can be impractical for human users. One of the main reasons is that, in most systems, the information the system can get from the user is limited to binary yes/no answers to (partial) membership queries, while the user might be able to provide more information that could aid the learning process [12].

In the literature on interactive CA, some works have explored the use of more *expressive* queries. In [26], the user is allowed to provide additional argumentation for their answer to a membership query. In the same spirit, LLMAcq [21] allows the user to give natural language feedback on why they answered negatively to a query. In [10, 16], *recommendation queries* are used, asking the user whether a specific constraint belongs to the target constraint model.

Finally, and most relevant to this paper, [5] introduced the *generalization query*, leveraging the idea that individual constraints often follow a larger pattern and can be grouped together. Such a query directly asks the user whether such a pattern, which can be translated to a *set of constraints*, belongs to the target model. For instance, in Sudoku, not-equal constraints follow patterns over rows, columns, and blocks. If a CA system can detect such patterns and query the user about them, a positive answer directly causes multiple constraints to be learned, each of which would otherwise require multiple membership queries. To this end, [5] proposed GenAcq, a method that generalizes using patterns over user-assigned variable types (e.g., row number in Sudoku). It asks generalization queries for combinations of variable types that can group together learned constraints ('should there be such a constraint between *all* sequences of variables of these types?').

Although this greatly reduces the number of queries, the existing GenAcq [5] method has some important limitations. First, the user has to manually assign relevant types to variables. While Mine&Ask [9] removes this requirement, by automatically extracting variable types, the type detection significantly increases the number of queries.Second, the type-based generalization queries have limited expressivity, and are not able to fully capture all patterns. Third, many of the patterns that can be formed based on variable types are patterns that do *not* exist in the actual model. Since no statistical likelihood measures are used, this can result in a large number of useless generalization queries.

In this paper, we improve on these limitations, by proposing a different approach to generalization, not based on variable types. We build on a recent method for *passive* learning, that can generalize complete instance-specific constraint models to different instances [27]. To achieve this, general patterns called constraint specifications (CSs) are extracted, which group together learned constraints based on a feature representation of these constraints. Importantly, these CSs are interpretable and more expressive than sequences of types. For this reason, we propose to use them during *interactive* CA, by introducing novel CS-based generalization queries. We then present an ML-based generalization query generation method, named GQ-Gen, and show how it can be used within any existing interactive CA algorithm. GQ-Gen trains an interpretable classifier using an incrementally growing dataset of constraints that have been learned or excluded during the acquisition process so far. It then extracts CSs from this classifier, and presents the relevant ones to the user for validation.

Specifically, our contributions are the following:

- We introduce the CS-based generalization query.
- We propose GQ-Gen, an ML-based approach to generate CS-based generalization queries and we show how to use it within any existing interactive CA algorithm.
- We experimentally show the effectiveness of our proposed methods.

- We perform a user-study to evaluate the interpretability of LLM-generated natural language formulations of the CS-based generalization queries we introduce.

2 Background

2.1 Constraint Satisfaction Problems

A *constraint satisfaction problem* (*CSP*) is a triple $P = (X, D, C)$, consisting of:

- a set of n variables $X = \{x_1, x_2, ..., x_n\}$, representing the entities of the problem,
- a set of n domains $D = \{D_1, D_2, ..., D_n\}$, where $D_i \subset \mathbb{Z}$ is the finite set of values for x_i,
- a constraint set $C = \{c_1, c_2, ..., c_t\}$.

A *constraint* c is a pair $(rel(c), var(c))$, where $var(c) \subseteq X$ is the *scope* of the constraint, and $rel(c)$ is a relation over the domains of the variables in $var(c)$, that (implicitly) specifies which of their value assignments are allowed. $|var(c)|$ is called the *arity* of the constraint. The constraint set $C[Y]$, where $Y \subseteq X$, denotes the set of constraints from C whose scope is a subset of Y. The set of solutions of a constraint set C is denoted by $sol(C)$. The goal in a CSP is to find a solution $s \in sol(C)$. An *example* e_Y is a (partial) assignment on a set of variables $Y \subseteq X$. e_Y is *rejected* by a constraint c iff $var(c) \subseteq Y$ and the projection $e_{var(c)}$ of e_Y on the variables in the scope $var(c)$ of the constraint is not in $rel(c)$ (i.e., e_Y violates c). A complete assignment e that is accepted by all constraints in C is a *solution* to C, i.e., $e \in sol(C)$. An assignment e_Y is called a *partial solution* iff $e_Y \in sol(C[Y])$. $\kappa_C(e_Y)$ represents the subset of constraints from $C[Y]$ that reject e_Y.

2.2 Interactive Constraint Acquisition

In interactive CA, the goal is to acquire the constraint set C of a CSP through a query-based interaction between the system and the user. The *vocabulary* (X, D) is assumed to be given by the user, along with a *language* Γ consisting of *fixed arity* constraints. Using the vocabulary (X, D) and the language Γ, the system generates the *constraint bias* B, which is the set of candidate constraints for the problem. The (unknown) target constraint set C_T is a constraint set such that for every example e it holds that $e \in sol(C_T)$ iff e is a solution to the problem the user has in mind. The goal of CA is to learn a constraint set C_L equivalent to the target constraint set C_T. A *(partial) membership query* AskMQ(e_Y), with $Y \subseteq X$, asks the user if a (partial) assignment e_Y is a (partial) solution or not, i.e., if $e_Y \in sol(C_T[Y])$. The acquisition process has *converged* on the learned set $C_L \subseteq B$ iff C_L agrees with the set of labeled examples E, and for every other such set $C \subseteq B$, it holds that $sol(C) = sol(C_L)$.

2.3 Type-based Generalization Queries

To reduce the number of queries required, [5] introduced the *generalization query*. Because we will introduce a different notion of generalization query, we henceforth refer to these generalization queries as *type-based generalization queries*. These queries aim to generalize a learned constraint c to sequences of types of variables that are part of the constraint. A *type* T is a subset of variables that share a common property, e.g., they lie in a specific row in Sudoku. A variable x is of type T iff $x \in T$. A tuple of variables $var = (x_1, \ldots, x_k)$ belongs to a sequence of types $s = (T_1, \ldots, T_k)$ (denoted by $var \in s$) iff $x_i \in T_i$ for all $i \in \{1, \ldots, k\}$. A sequence $s' = (T'_1, T'_2, \ldots, T'_k)$ covers another sequence $s = (T_1, T_2, \ldots, T_k)$ (denoted by $s \sqsubseteq s'$) iff $T_i \subseteq T'_i$ for all $i \in \{1, \ldots, k\}$. A relation r holds on a sequence of types s iff

$(r, var) \in C_T$ for all $var \in s$. A type-based generalization query $AskGen(s, r)$ is asking the user if a relation r holds on a sequence of types s. The query $AskGen(s, r)$ is answered positively iff the relation r holds for every sequence var of variables covered by s.

The GenAcq [5] system was proposed to allow interactive CA to use type-based generalization queries. Given a learned constraint c, and the set of variable types $\mathcal{T}$, GenAcq tries to generalize the relation $rel(c)$ to all sequences of type S that cover the variables in its scope $var(c)$, except ones that involve constraints that have been removed from B, or that have already been answered negatively. It returns a (potentially empty) set $\mathcal{G}$, consisting of accepted sequences of types.

2.4 Constraint Specifications

In this work, we will introduce a new type of generalization query, based on *constraint specifications (CSs)*. CSs were formalized in [27], based on the insight that constraint problems typically do not consist of just sets of individual constraints, but of higher-level requirements that define entire sets of constraints. A CS aims to represent such a higher-level requirement of the problem, and can be flattened into a set of individual constraints.

▶ **Definition 1.** *A* constraint specification *(CS) is a triple (r, P, Q), defining*

- *a relation r*
- *a variable partition function P that maps the set of variables X onto a partition based on certain characteristics*
- *a set of sequence conditions Q that specify to which sequences of variables in each partition the constraints should be applied*

The relation r determines the relation of the constraints that the CS specifies (e.g., $\neq$). The partitioning function P specifies a way to partition the variables (e.g., into rows or columns). Finally, the sequence conditions Q specify which scopes within the variable partitions the constraints apply to (e.g., only ones at even indices). Given a CS, the following generator template can be used to generate the corresponding constraints:

```
Foreach Y ∈ P(X):
  Foreach sq ∈ combinations(Y, arity(r), Q):
    c ← (rel(c) = r, var(c) = sq)
```

Example 2 in Section 3 gives an example of an instantiated CS generator.

A constraint c is covered by a CS $s = (r, G, Q)$ (denoted by $c \sqsubseteq s$) iff $rel(c) = r$ and there exists a $p_i \in P(X)$ for which $v \in p_i$ for all $v \in var(c)$, and $Q(var(c))$ is true. A CS s is covered by another CS s' (denoted by $s \sqsubseteq s'$) iff $c \sqsubseteq s'$ for all $c \sqsubseteq s$.

2.5 Use of statistical ML in CA

Recent work has explored using statistical ML in CA in different ways [30, 27]. To enable this, a set of known constraints C is used to build a training set E. Each training example in E is a tuple $(\mathbf{x}_i, y_i)$, corresponding to a constraint $c_i \in C$. For each tuple $(\mathbf{x}_i, y_i)$, $\mathbf{x}_i = \phi(c_i)$ is a feature vector for constraint c_i, and $y_i = [c_i \in C_T]$ a Boolean label expressing whether constraint c_i is part of C_T or not. Thus, $E = \{(\phi(c_i), [c_i \in C_T]) \mid c_i \in C\}$. This dataset is used to train a classifier $\mathcal{M}$ to learn a function f that can classify constraints as part of C_T or not. Such a classifier has been used in two distinct ways. First, [30] proposed an interactive framework for using the classifier to guide classic membership query generation. Second, the classifier has been used in passive acquisition for across-instance generalization [27]. We will show that classifier-based generalization can also be integrated into an interactive framework.

3 Revisiting Generalization Queries

We start by revisiting generalization queries in interactive CA, and we introduce CS-based generalization queries. Generalization queries are based on the idea that individual constraints often follow larger patterns, based on the requirements of the problem, and can thus be grouped together. For example, in Sudoku, the constraints are not defined individually, but through patterns based on the requirements that "all rows/columns/blocks must contain only distinct values".

As explained in Section 2.3, a type-based generalization query $AskGen(s, r)$ from GenAcq [5] asks the user whether a relation r holds on a sequence of types s. If the answer is "yes", a constraint with relation r is learned for each tuple of variables covered by these types. However, this approach suffers from several limitations, which we illustrate with Example 1.

▶ **Example 1.** Assume we use CA to acquire the constraints of Sudoku, and we have learned the constraint $cell_{1,1} \neq cell_{1,9}$, i.e., the first cell of the first row must have a different value than the last cell of the first row. GenAcq will now try to generalize this constraint. Assume that each variable has been assigned three types by the user, one for its specific row, column, and block. GenAcq will now attempt to generalize over all combinations of types of the two variables, i.e., (row_1, row_1), (row_1, col_9), (row_1, $block_3$), (col_1, row_1), (col_1, col_9), (col_1, $block_3$), ($block_1$, row_1), ($block_1$, col_9), ($block_1$, $block_3$). After asking these nine generalization queries, only combination (row_1, row_1) will have been answered positively by the user, expressing that all variables in the first row need to have distinct values.

One limitation shown in the example is that the system separately asks each generalization query, without making an assessment of which generalized pattern is likely to hold. We will improve on this using statistical ML in Section 4.

Another limitation is the limited expressivity of the type-based generalization queries. In this example, they can only generalize over each individual row (or column or block). Fewer queries would be required if the system were able to generalize directly to the full problem requirement "in each row, all variables must have distinct values". This kind of general requirement can be captured by *Constraint Specifications* (CS), as shown in Example 2:

▶ **Example 2.** Consider the problem from Example 1. To generalize to the actual problem requirement "all variables in the same row must have a different value", we would need to generalize to the following CS:

```
Foreach row ∈ all_rows:
    Foreach cell1, cell2 ∈ all_pairs(row):
        c ← cell1 ≠ cell2
```

with `all_pairs(row)` == `combinations(row, 2, ∅)`. This CS partitions the variables in rows, expressing that all pairs of variables in each row are subject to $\neq$ constraints.

This does not require specifying variable types: only generic constraint features, that can then be used to derive partition functions and sequence conditions, are needed, like the ones from [30, 27]. Through its partitioning function, a CS can express that not-equals constraints exist between all pairs of variables in *each* row, instead of grouping only on a single row. Additionally, through its sequence conditions, a CS can capture much more complex patterns than a sequence of variable types can.

We now define a CS-based generalization query:

▶ **Definition 2.** *A CS-based generalization query AskGen(s) asks the user whether a CS s holds for their problem. A positive answer implies that* $\{c \mid c \sqsubseteq s\} \subseteq C_L$*, whereas a negative answer implies that* $\{c \mid c \sqsubseteq s\} \not\subseteq C_L$.

3.1 Generating generalization queries in natural language.

To ease the interpretation of generalization queries, especially for users that are not used to reading pseudocode, we show how to generate natural language formulations of the queries. To this end, we use general-purpose LLMs, which have previously been used in CP to translate from natural language to CP model code [22]. We use them here in the opposite direction: given the CS, generate a natural language description, which will then be posed as a query.

We use the well-established methodology of *prompting* [20], where we provide as input to the LLM a natural language instruction to translate the given CS into a natural language query. The prompt includes the CS; a description of the variables of the problem, along with their structure and names as given by the user to the CA system; the instruction to translate the CS by grounding it in the problem description; as well as 1 to 6 in-context examples of relation/partitioning/sequence features and their meaning in natural language. The full prompt and examples of generated natural language queries in our experiments are provided in Appendix C. For the CS of Example 2, this approach results in the generated natural language query *"In each row, should every pair of cells have distinct values?"*

4 Interactive CA with CS-based Generalization Queries

Just as the existing GenAcq method is hindered by the need to pose generalization queries for all combinations of types, the use of CSs would become inefficient if a separate query had to be generated and posed for every possible CS that generalizes a learned constraint.

We instead propose to leverage the pattern-matching capabilities of statistical ML by directly extracting CSs from a learned ML model. This has the following advantages:

- It leverages data-driven ML to focus on the patterns that are most relevant according to statistical correlations.
- It does not require user-provided variable types. Instead, the classifier identifies structure in the data through generic constraint features, and can group constraints in different ways to derive suitable CSs.

Although an inductive bias remains based on the constraints' feature representation, our experiments show that this representation does not need to be problem-specific and can be extensive. The ML component can then select the statistically most promising patterns.

In this section, we first show how we can generate these queries, and then how any existing interactive CA framework can be extended with CS-based generalization queries, followed by LLM-based translation into natural language queries.

4.1 Generation of Generalization Queries from Classifiers

In this section, we assume the availability of a trained classifier that can classify constraints as belonging to C_T, as described in Section 2.5. In Section 4.2, we will describe how such a classifier can be trained within the interactive CA loop.

Our proposed method for generating CS-based generalization queries, named GQ-Gen, makes use of the ExtractCSs method from [27] to extract CSs from decision rules. To do so, one needs to use ML classifiers from which if-then decision rules can be extracted, such as decision trees or rule-based classifiers. The if-then rules that lead to a positive classification contain the conditions for a constraint to be part of C, expressed in terms of the feature representation $\phi(c)$. Each positive rule can then be converted to a CS. To enable this, the feature representation $\phi(c)$ has to be divided into three categories that correspond

Algorithm 1 GQ-Gen$(C_L, C_-, NegativeQ, \mathcal{M})$

Input: $C_L, C_-, NegativeQ, \mathcal{M}$
Output: S_p : a set of CSs
1: $S_p \leftarrow$ ExtractCSs $(\mathcal{M})$
2: $S_p \leftarrow \{s \in S_p \mid \nexists s' \in NegativeQ \wedge s' \sqsubseteq s\}$
3: $S_p \leftarrow \{s \in S_p \mid \nexists c \in C_- \mid c \sqsubseteq s\}$
4: $S_p \leftarrow \{s \in S_p \mid \exists c \notin C_L \mid c \sqsubseteq s\}$
5: **return** S_p

to the elements of CSs: *Relation*, *Partitioning*, and *Conditioning* features. Such features are constructed from a *language* of generic relation, partition and sequence condition templates.

- *Relation features.* Capture properties of the relation of the given constraint. For example, which relation from Γ is used, e.g. $=, \neq, \leq$ etc.
- *Partitioning features.* Capture whether the variables in the scope of constraint c have characteristics in common that can be used to partition them. For example, when the variables are part of an n-dimensional array of decision variables, if they are in the same row, column, or in general, if they have the same index in one dimension.
- *Conditioning features.* Features describing how the variables in the scope of the constraint relate in different ways, e.g., what the distance between their indices in an array is, what the smallest or largest index is in a dimension, etc.

The full list of features that we use in our experiments is given in Appendix A.

To construct a CS, the conditions of a given decision rule are considered one by one. Based on the features they involve and the groups they belong to, the elements of the CS are constructed. We treat this method of extracting CSs from a classifier as a function

$$\text{ExtractCSs} : \mathcal{H} \rightarrow \mathcal{P}(\mathcal{S}), \tag{1}$$

where $\mathcal{H}$ is the space of classifiers that support decision rule extraction, and $\mathcal{S}$ is the set of possible CSs. $\mathcal{P}(\mathcal{S})$ refers to the powerset of $\mathcal{S}$. Given a classifier $\mathcal{M} \in \mathcal{H}$, ExtractCSs$(\mathcal{M})$ returns a subset of $\mathcal{S}$ with the extracted CSs: ExtractCSs$(\mathcal{M}) = \{S_1, S_2, \ldots, S_k\} \subseteq \mathcal{S}$.

However, we do not directly use all these CSs, but filter them as shown in Algorithm 1. Concretely, in an interactive CA setting, we have to filter out three sorts of CSs:

- Some of the CSs might have already been asked and rejected by the user, or cover a CS that has already been rejected. We hence need to keep track of the set of all queries that have been negatively answered by the user ($NegativeQ$) and filter these out (line 2);
- If a CS covers a previously rejected constraint, then the CS can never be a true pattern. We hence need to keep track of all constraints rejected by the user C_-, and filter out the CSs that cover any of these constraints (line 3);
- Finally, we only consider CSs that cover at least one candidate constraint that has not been learned yet (line 4).

4.2 Classifier-based Generalization Queries for Interactive CA

As discussed in Section 2.5, recent work has proposed a framework for interactive CA that guides the generation of standard membership queries using ML classifiers [30]. In this framework, a dataset E of constraints is grown incrementally during the acquisition process.

Algorithm 2 ML-based interactive CA with generalization queries, with our contribution in blue.

Input: C_{in}, C_{excl}, B, X
Output: C_L, C_-: the learned set of constraints

1: $C_L \leftarrow C_{in}, \quad C_- \leftarrow C_{excl}$
2: $E \leftarrow \{(\phi(c_i), [c_i \in C_L]) \mid c_i \in C_L \cup C_-\}$
3: $NegativeQ \leftarrow \emptyset$
4: **while** True **do**
—— Learning from Membership Queries ——
5: $\mathcal{M} \leftarrow$ TrainClassifier(E)
6: $e, Y \leftarrow$ MQ-Gen(C_L, B, $\mathcal{M}$)
7: **if** $e =$ nil **then return** C_L, C_- ▷ Converged
8: **if** AskMQ(e_Y) = True **then**
9: $C_- \leftarrow C_- \cup \kappa_B(e_Y)$
10: $E \leftarrow E \cup \{(\phi(c_i), False) \mid c_i \in \kappa_B(e_Y)\}$
11: $B \leftarrow B \setminus \kappa_B(e_Y)$
12: **else**
13: $S \leftarrow$ FindScope(e, Y, B, C_-, E, $\mathcal{M}$)
14: $C \leftarrow$ FindC(S, C_L, B, C_-, E, $\mathcal{M}$)
15: $C_L \leftarrow C_L \cup C$
16: $E \leftarrow E \cup \{(\phi(c_i), True) \mid c_i \in C\}$
—— Learning from Generalization Queries ——
17: $\mathcal{M} \leftarrow$ TrainClassifier(E)
18: $S_p \leftarrow$ GQ-Gen $(C_L, C_-, \mathcal{M}, NegativeQ)$
19: **for** $s \in S_p$ **do**
20: **if** AskGQ(s) $= True$ **then**
21: $C_L \leftarrow C_L \cup \{c \mid c \sqsubseteq s \wedge var(c) \subseteq X\}$
22: $E \leftarrow E \cup \{(\phi(c_i), True) \mid c_i \sqsubseteq s\}$
23: $S_p \leftarrow S_p \setminus \{s' \in S_p \mid s' \sqsubseteq s\}$
24: **else**
25: $NegativeQ \leftarrow NegativeQ \cup \{s\}$
26: $S_p \leftarrow S_p \setminus \{s' \in S_p \mid s \sqsubseteq s'\}$

Whenever a constraint is removed from the bias B, it is added to E with a negative label, and whenever a constraint is added to C_L, it is added to E with a positive label. An ML classifier is trained on this dataset during the CA process, and its probabilistic predictions for the remaining candidates are used to guide membership query generation with the aim of minimizing the required number of queries.

We now posit that this classifier can also be used to generate CS-based generalization queries, using the above GQ-Gen method. This means that we can add our generalization queries to any interactive CA method, if it is modified to train a rule-based classifier over the learned/rejected constraints.

We show this for an ML-guided version of the QuAcq method in Algorithm 2. Lines 5-16 correspond to the standard ML-based framework from [30], and lines 17-26 are our contribution. In the top-level loop, after asking a membership query, the system enters the generalization phase. Here, GQ-Gen is invoked (line 18) to generate CS-based generalization queries, which are collected in set S_p. These queries are then posted to the user for validation (lines 19-20). If a candidate CS s is accepted (lines 21-23), the constraints covered by s are

added to C_L, E is updated with these constraints, and all CSs covered by s are removed from S_p. If s is rejected (lines 25-26), all CSs covering s are removed, and s is recorded in $NegativeQ$ to avoid future redundant queries. When all CSs in S_p have been posted, the main loop starts over (line 4). The convergence criterion (line 7) is unmodified.

Notice that we use GQ-Gen after each generated membership query, even if no new constraint was learned. This is done because also positive queries change E (line 10) and can hence lead to a different trained model and thus new CSs.

▶ **Example 3.** Consider again the Sudoku setting from Example 1. Assume that, during acquisition, one or more constraints between cells in the same row have already been learned, such as $cell_{1,1} \neq cell_{1,9}$ and $cell_{1,2} \neq cell_{1,5}$. These constraints are added to E with a positive label, while excluded constraints, such as not-equals constraints between cells that do not share a row, column, or block, are added with a negative label.

A rule-based classifier trained on E can then identify that constraints whose scope variables belong to the same row are likely to be positive. For example, it may learn the following positive-classification rule:

```
Relation == "!=" & Dim0_same == "true" then "target constraint"
```

Here, `Relation` is a relation feature, while `Dim0_same` is a partitioning feature indicating that the two scope variables have the same index in the first dimension, i.e., they belong to the same row. Following the extraction procedure of ExtractCSs, `Relation == "!="` determines the relation of the CS, while `Dim0_same == "true"` determines that constraints should be generated within each group of variables sharing the same first-dimension index.

Then, instead of separately asking type-based queries such as (row_1, row_1), (row_2, row_2), and so on, GQ-Gen extracts the following CS from the classifier:

```
Foreach row ∈ all_rows:
    Foreach cell1, cell2 ∈ all_pairs(row):
        c ← cell1 ≠ cell2
```

This CS is posed to the user as a single generalization query, for instance in natural language as: *"In each row, should every pair of cells have distinct values?"* The exact procedure for extracting such CSs from decision rules is described in [27].

After a positive answer, all constraints covered by the CS are added to C_L and to E with a positive label. Thus, a single accepted CS-based generalization query can learn the row-wise Sudoku constraints at once. The same mechanism can later extract analogous CSs for columns and blocks, once the learned and rejected constraints provide sufficient statistical evidence for these patterns.

In this updated framework, the role of the ML classifier is more pronounced than in earlier works. Previously, when an ML component was incorporated, its probabilistic constraint-level predictions were used to guide the generation of membership queries, in order to minimize the expected number of queries to converge. The new framework gives the ML classifier a more central role. It does not only utilize the classifier's constraint-level predictions, but also directly extracts the structure it has implicitly learned, and presents this to the user for verification. Hence, rather than acquiring all constraints individually through a (guided) exhaustive search procedure, the system now only needs to acquire enough constraints through membership queries to enable effective pattern recognition, when this is possible.

5 Experimental Evaluation

Our experimental evaluation aims to answer the following questions:

(Q1) How does the use of GQ-Gen for CS-based generalization queries perform?
(Q2) How does the number of asked queries grow as problem size increases?
(Q3) To what extent can users interpret and answer LLM-generated natural language formulations of generalization queries?

5.1 Experiment setup

5.1.1 Benchmarks

We used 9 benchmarks in our experiments, selected to include different types of constraint specifications (CSs). In particular, we consider both fully structured problems and problems that include additional constraints that do not follow a general pattern, allowing us to evaluate the behavior of our methods when certain constraints cannot be generalized. Specifically, we used the following:

Fully structured benchmarks:

Latin Square Latin square of order n is an $n \times n$ grid where each row and each column contains each value from 1 to n exactly once. The variables form an $n \times n$ matrix, with domains $D_i = \{1, \ldots, n\}$. We used an instance with $n = 10$, giving a vocabulary of 100 variables in a 10×10 matrix and domains of size 10. C_T has 900 $\neq$ constraints. The bias was built with the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$, resulting in $|B| = 29,700$ binary constraints.

Sudoku The Sudoku puzzle is a $n^2 \times n^2$ grid, which must be completed in such a way that all the rows, columns, and n^2 non-overlapping $n \times n$ squares contain the distinct numbers. We used an instance with $n = 3$, leading to a grid size of 9×9. This gives a *vocabulary* having 81 variables, in a 9x9 matrix, and domains of size 9. The target constraint network consists of 810 $\neq$ constraints. The bias was initialized with 19,440 binary constraints, using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$.

Exam Timetabling There are ns semesters, each containing cps courses. We want to schedule the exams of the courses in a period of d days, with each day having t slots available for exams. The variables are the courses ($|X| = ns \cdot cps$), having as domains the timeslots they can be assigned on ($D_i = 1, ..., t \cdot d$). There are $\neq$ constraints between each pair of exams, because they cannot be assigned in the same slot of a day. Also, two courses in the same semester cannot be examined on the same day, which is expressed by the constraints $\lfloor x_i/spd \rfloor \neq \lfloor x_j/spd \rfloor$, $\forall i, j$ in the same program. We used an instance with $ns = 9$, $cps = 6$, $d = 14$ and $t = 9$. This resulted in a model with 54 variables and domains of size 126. C_T consists of 1,566 constraints. The language given is $\Gamma = \{\geq, \leq, <, >, \neq, =, \lfloor x_i/spd \rfloor \neq \lfloor x_j/spd \rfloor\}$, creating a bias of size 10,017.

Nurse rostering There are n nurses, s shifts per day, ns nurses per shift, and d days. The goal is to assign a nurse to all existing shifts. In the model, the variables represent the shifts, and we have a total of $d \cdot s \cdot ns$ shifts. The variables are modeled in a 3D matrix. The domains of the variables are the nurses ($D^{x_i} = \{1, ..., n\}$). We have the following 2 requirements: Each shift in a day must be assigned to a different nurse and the last shift of a day must be assigned to a different nurse than the first shift of the next day. In the instance used in the experiments, we have $d = 7$, $s = 3$, $ns = 5$, and $n = 18$. This results in $|X| = 105$ with domains $\{1, ..., 18\}$. C_T consists of 885 $\neq$ constraints. The bias was built using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$, resulting in $|B| = 32,760$.

Distance-based Grid Coloring We consider a grid of $r \times c$ cells. Each cell is assigned a color from $\{1, \ldots, k\}$. Any two cells within distance d (under a given metric) must have different colors. Variables are the cells in an $r \times c$ matrix, with domains $D_i = \{1, \ldots, k\}$. The target network consists of $\neq$ constraints between all pairs of cells within distance d in

rows and columns (Chebyshev distance). We used $r = 10$, $c = 10$, $k = 18$, $d = 2$. This yields 100 variables, domains of size 18, and C_T with 918 $\neq$ constraints. The bias uses $\Gamma = \{\geq, \leq, <, >, \neq, =\}$, giving $|B| = 29,700$ binary constraints.

Benchmarks including additional constraints:

Purdey A small logic-grid puzzle with 3 entities and 3 categories (drink, store, pet). Each category has 3 items, each assigned to one entity. Variables form a 3×3 matrix, with domains $\{1, 2, 3\}$. The target network enforces $\neq$ constraints in all pairs of variables on each row, and includes additional equality, inequality, and adjacency constraints. C_T has 13 constraints. The bias uses $\Gamma = \{\geq, \leq, <, >, \neq, =, |x_i - x_j| = 1\}$, giving $|B| = 252$ binary constraints.

Murder A logic puzzle with 5 suspects, 5 motives, 5 evidence objects, and 5 activities. The goal is to assign each suspect a motive, object, and activity under given clues. Variables are arranged in a 4×5 matrix (4 categories, 5 options), with domains $\{1, \ldots, 5\}$. The target network includes $\neq$ in each pair of variables on each row, and 12 additional equality and inequality constraints from some given clues. The instance has 20 variables and domains of size 5. C_T has 52 constraints. The bias uses $\Gamma = \{\geq, \leq, <, >, \neq, =\}$, giving $|B| = 1,140$ binary constraints.

Zebra A logic-grid puzzle with 5 houses and 5 categories (color, nationality, drink, smoke, pet). Each category has 5 items, each assigned to one house. Variables form a 5×5 matrix (one row per category), with domains $\{1, \ldots, 5\}$. The target network enforces AllDifferent on each row and includes equality, inequality, and adjacency constraints (e.g. $|x_i - x_j| = 1$). C_T has 61 constraints. The bias uses $\Gamma = \{\geq, \leq, <, >, \neq, =, x_i{+}1 = x_j, x_j{+}1 = x_i, |x_i{-}x_j| = 1\}$, resulting in $|B| = 2,700$ binary constraints.

Greater-than Sudoku The Greater-than Sudoku is a variant of Sudoku in which additional $>$ and $<$ are included between some adjacent cells. The instance we used has a grid size of 9×9, consisting of 81 variables with domains of size 9. The target network contains 810 binary $\neq$ constraints on rows, columns, and shapes, and 24 additional $>, <$ constraints. The bias B was constructed using the language $\Gamma = \{\geq, \leq, <, >, \neq, =\}$ and contains 19,440 binary constraints.

5.1.2 Comparison

Metrics: The comparison is based on *the total number of queries*. We also present the *number of generalization queries*, as they can potentially be more difficult to answer. We do not present results regarding the time to generate the generalization queries, as in our experiments this never exceeded 3 seconds. The results presented are the average over 20 runs. For statistical comparison, we apply a *paired Wilcoxon signed-rank test*, with a significance level of $\alpha = 0.05$. In the tables, the best result is highlighted in **bold**.

Algorithms: We compare our method with state-of-the-art algorithms: QuAcq [4], QuAcq with GenAcq [5], and QuAcq with Mine&Ask [9]. In all algorithms, we used the following components:

- The PQ-Gen method was used for query generation. PQ-Gen takes 2 parameters, i.e., a limit on the bias constraints for enabling the optimization step, and a time limit for optimization. We used 10,000 as the bias limit, and 1 second as the time limit.
- The ML framework for guiding membership queries from [30] was used.
- For the FindScope component, we used FindScope-2 [4].
- For the FindC component, we used the function from [6].

- A decision tree was used as the classifier in the ML component, based on the results from [27]. The decision tree was implemented in scikit-learn [24]. It was used with max_depth = 7, which was tuned on the benchmarks used to minimize the number of queries. The default values were used for the rest of the parameters.

GenAcq was used with the best settings reported in [5]. That is, a cutoff of 3 consecutive negative generalization queries was enforced, and the heuristic min_var was used to order the sequences of types to ask. Mine&Ask was used with optimizing modularity as the mining strategy, which presented the best results in [9]. The authors do not report the cutoff used in their experiments, and thus after some experimentation we set it to 10.

Constraint feature representation: Our feature representation is based on the language Γ of the constraint relations and a language of partitioning functions and sequence conditions that can be used to construct CSs. Partitioning features are derived from the dimensions of the tensor in which the variables are defined, as well as from *latent dimensions* discovered through the divisors of each dimension size, following [27]. For each (latent) dimension, we include a partitioning feature indicating whether all variables in the constraint's scope share the same index in that dimension. Additionally, we use numerical conditioning features as proposed in [30], including the average, maximum, and minimum variable indices in each (latent) dimension, as well as their average difference and absolute average difference. Further details about the feature representation are provided in Appendix A.

Experiment settings: All the experiments were conducted on an Ubuntu VM with 24 vCPUs and 16GB of RAM, hosted on a machine equipped with an Intel(R) Xeon(R) CPU E5-2630 v3, with 2.40GHz clock speed, and 16 GB of RAM. All methods and benchmarks were implemented[1] in Python, using the PyConA CA library [28], which uses the CPMpy constraint programming and modeling library [15]. OR-Tools' CP-SAT [25] solver was used for query generation, interfaced through CPMpy. The decision tree classifiers were implemented using the Scikit-Learn library [24].

5.2 Results

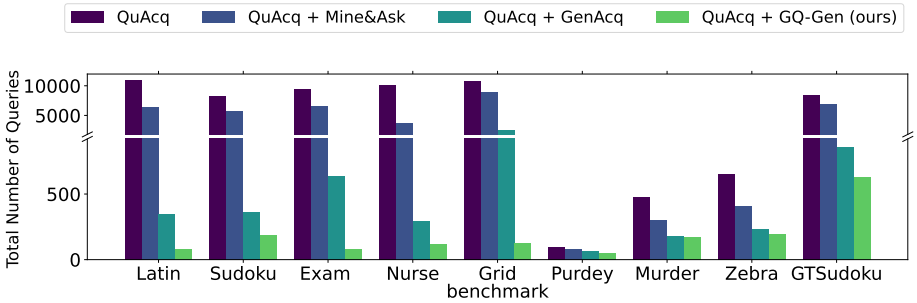
Q1: Performance of GQ-Gen

To evaluate the performance of GQ-Gen, we use it inside QuAcq, and compare it with the use of the state-of-the-art type-based generalization approaches Mine&Ask and GenAcq, as well as pure QuAcq without generalization. The results are shown in Figure 1.

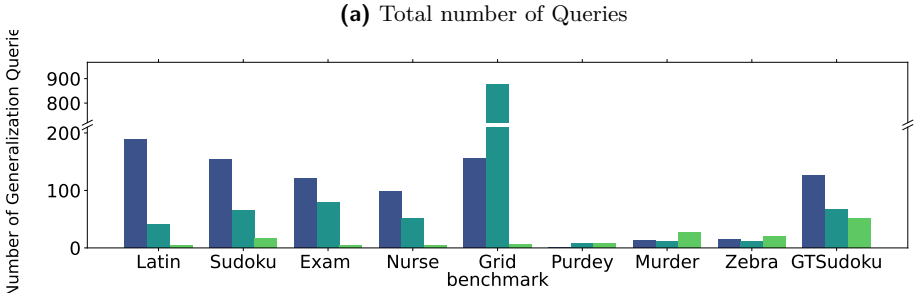
We can see that the use of any kind of generalization query (type-based in GenAcq and Mine&Ask, CS-based in GQ-Gen) reduces the number of queries. However, for type-based generalization, when variable types are not given and need to be detected using Mine&Ask, the reduction is far less significant than when types are explicitly given to GenAcq. In contrast, our proposed ML-based approach GQ-Gen substantially reduces the number of queries without requiring the user to manually define problem-specific variable types.

Importantly, across all benchmarks, GQ-Gen consistently results in a lower number of queries compared to GenAcq. In all fully structured benchmarks (i.e., Latin, Sudoku, Exam, Nurse, Grid) the improvement is noticeably higher: 78% in Latin, 48% in Sudoku, 87% in Exam, 60% in Nurse, and 95% in Grid. The huge reduction in the number of queries in Grid is due to the limitation of type-based generalization queries, which capture patterns for each combination of types separately.

[1] Our code will be made available online upon acceptance.



(a) Total number of Queries



(b) Number of Generalization Queries

Figure 1 Performance of state-of-the-art and our proposed approach GQ-Gen

The reduction compared to GenAcq is still significant, but smaller, in benchmarks that contain constraints not following any pattern. This reduction comes specifically from the queries needed to learn the part of the problem that follows a pattern, while not affecting the performance of CA when learning the additional constraints. The smallest decrease is present in Murder (6%), as in this problem the number of additional constraints is similar to the amount of constraints learned through generalization. On the other hand, in Zebra and Purdey, GQ-Gen needs 17% fewer queries, while we observe that in GTSudoku which is the largest such problem the reduction reaches 26%. Focusing on Figure 1(b), we can see that, in most problems, the use of GQ-Gen also significantly reduces the number of generalization queries. This is because CS-based generalization queries are more expressive than type-based ones. The only benchmarks in which this does not happen are the 3 logic grid puzzles, because the number of generalization queries is already very small.

Q2: Performance of our methods on different problem sizes

We now evaluate how our proposed methods perform as problem size increases. We focus on the Exam, Nurse, and Grid benchmarks, which can be generated at different sizes through their parameters, and most closely resemble real-world problems. For each problem class, we generated 10 instances of increasing size. We include QuAcq as a baseline when no generalization is used, QuAcq + GenAcq as the baseline using generalization queries, and our proposed approach QuAcq + GQ-Gen. Table 1 presents the results.

We can directly observe that when generalization is not used, the number of queries increases quickly with increasing problem size. The use of generalization shows significant improvements even in small instances, with the difference in performance becoming larger as the problem size increases. Notably, on the largest instances, using QuAcq with GQ-Gen (our proposed approach) reduces the number of queries by up to 99% compared to QuAcq.

Comparing the use of GQ-Gen against GenAcq, we see that our proposed approach

Table 1 Evaluating the effect of the problem size.

	All Queries, Exam Timetabling									
Nr. Constraints:	*77*	*143*	*218*	*348*	*488*	*716*	*959*	*1317*	*1699*	*2231*
QuAcq	377	734	1248	2046	2916	4338	5935	7994	10289	13108
QuAcq + GenAcq	101	149	191	233	385	374	542	557	818	826
QuAcq + GQ-Gen	**32**	**35**	**37**	**37**	**46**	**60**	**52**	**79**	**66**	**73**
	All Queries, Nurse Rostering									
Nr. Constraints:	*53*	*171*	*216*	*394*	*476*	*640*	*722*	*804*	*1275*	*1854*
QuAcq	403	1671	2338	4210	5368	8022	9273	10546	>15K	>15K
QuAcq + GenAcq	75	125	156	182	222	308	356	394	423	442
QuAcq + GQ-Gen	**45**	**115**	**100**	**122**	**232**	**93**	**91**	**202**	**286**	**228**
	All Queries, Grid Coloring									
Nr. Constraints:	*168*	*270*	*396*	*465*	*546*	*627*	*708*	*813*	*918*	*1140*
QuAcq	1229	2359	3831	4621	5628	6702	7868	9325	10852	13828
QuAcq + GenAcq	343	600	930	1138	1349	1494	1751	2017	2305	5016
QuAcq + GQ-Gen	**92**	**162**	**114**	**138**	**215**	**136**	**150**	**119**	**189**	**148**

scales much better to larger problem sizes. The number of queries is substantially reduced, with GQ-Gen needing up to 91% less queries in the largest instance of Exam, 49% less queries in the largest instance of Nurse, and up to 97% in Grid in which type-based generalization queries cannot capture the pattern well. Interestingly, the decrease in the number of generalization queries when GQ-Gen is used, as reported in Appendix B, gets larger when the size of the problem increases. This is due to the low expressivity of the type-based generalization queries of GenAcq. The larger the instance, the more variable types there are (more rows, columns, semesters, etc.), and the more type-based generalization queries GenAcq needs to find all patterns. GQ-Gen does not suffer from this limitation.

Q3: Interpretability

To evaluate the *interpretability* of CS-based generalization queries by human users, we conducted a small-scale user study. In particular, we assessed whether the LLM-generated natural language formulations of these queries, as described in Section 3.1, are understandable and answerable for users. To this end, we evaluated four indicators of interpretability:

1. the *clarity* of the query as perceived by users,
2. the *correctness* of the answers provided by participants,
3. the *confidence* users report in their answers
4. the *time required* to understand and answer each query.

High clarity ratings, high correctness and confidence, and short response times indicate that the generated queries are easy to interpret.

A total of 19 participants[2], recruited from a university's researchers (not including any authors), completed the evaluation via a Google Forms survey. Participants were provided

[2] Participation in the study was voluntary and anonymous. No personal or sensitive data were collected. According to our institutions' guidelines, this type of evaluation did not require formal ethical approval.

Table 2 Summary of user evaluation responses.

Clarity (1-5)					Answer		Confidence (1-3)			Time Spent		
1	2	3	4	5	Incorrect	Correct	1	2	3	<30s	30s–1min	>1min
3%	7%	12%	26%	53%	11%	89%	9%	22%	69%	70%	25%	5%

with task instructions and a short description of the domain relevant to each benchmark, and then answered each query and evaluated its clarity, their confidence, and the time required to respond. For this study, we collected all such queries that were asked in the experiment of Q1 during one run, focusing on the benchmarks Sudoku, GTSudoku, Exam, and Nurse. Each query was automatically translated into natural language using an LLM. We used gpt-4.1 for this task. Our prompt, and additional details for the user study, can be found at Appendix C

Averaged results for all users on all queries are presented in Table 2. The results indicate that the natural language generalization queries produced by our system were generally clear and understandable. Over half of the responses (53%) assigned the maximum clarity score (5), while 26% gave a score of 4, suggesting that a strong majority found the queries easy to interpret. In addition, 89% of responses were correct, indicating that both the LLM generation and the participant's interpretation were mostly correct. Confidence levels were also high, with 69% of answers having the highest confidence level, and only 9% having the lowest confidence. Importantly, most participants were able to understand and answer most queries in under 30 seconds (70%), and only 5% required more than one minute. These results suggest that CS-based generalization queries can be made interpretable through LLM-based natural language translation.

6 Conclusions

We focused on a key limitation of interactive CA: the large number of queries. We revisited the use of generalization queries, introducing CS-based generalization queries. They offer more expressive generalization than existing type-based generalization, without requiring explicit variable types given by the user. We then proposed an ML-based approach to generate such generalization queries by extracting CSs using an ML model. Our experiments show that our approaches are highly efficient. A small-scale user study illustrated the interpretability of natural language translations of our queries.

With our enhanced ML-based framework for interactive CA, the ML classifier gets a more central role. In previous works, even when statistical ML was employed, it was only used to provide constraint-level predictions to guide the search-based procedure. Our work changes the dynamic between search-based learning and statistical ML by extracting learned patterns directly from the ML model as CSs, which are then used as queries.

Future work can explore incorporating techniques from active learning, guiding the interaction with the user to parts of the problem that the ML model is most uncertain about, aiming to improve generalization. Moreover, ML-based generalization can enable learning classes of constraints that are currently challenging due to the explosion of the candidate space, such as global constraints. Finally, as interaction with human users can involve incorrect answers, especially in cases that LLMs are used to translate queries to natural language, there is a need for robust CA methods. The tighter integration with statistical ML can be a stepping stone towards robust interactive CA systems.

References

1 Hugo Barral, Mohamed Gaha, Amira Dems, Alain Côté, Franklin Nguewouo, and Quentin Cappart. Acquiring constraints for a non-linear transmission maintenance scheduling problem. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2024.

2 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *Principles and practice of constraint programming*, pages 141–157. Springer, 2012.

3 Senne Berden, Mohit Kumar, Samuel Kolb, and Tias Guns. Learning max-sat models from examples using genetic algorithms and knowledge compilation. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, 2022.

4 Christian Bessiere, Clement Carbonnel, Anton Dries, Emmanuel Hebrard, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, Kostas Stergiou, Dimosthenis C Tsouros, and Toby Walsh. Learning constraints through partial queries. *Artificial Intelligence*, 319:103896, 2023.

5 Christian Bessiere, Remi Coletta, Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, and El-Houssine Bouyakhf. Boosting constraint acquisition via generalization queries. In *ECAI*, pages 99–104, 2014.

6 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, Toby Walsh, et al. Constraint acquisition via partial queries. In *IJCAI*, volume 13, pages 475–481, 2013.

7 Christian Bessiere, Remi Coletta, Barry O'Sullivan, Mathias Paulin, et al. Query-driven constraint acquisition. In *IJCAI*, volume 7, pages 50–55, 2007.

8 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

9 Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, Christian Bessiere, and El Houssine Bouyakhf. Detecting types of variables for generalization in constraint acquisition. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 413–420. IEEE, 2015.

10 Abderrazak Daoudi, Younes Mechqrane, Christian Bessiere, Nadjib Lazaar, and El Houssine Bouyakhf. Constraint acquisition using recommendation queries. In *IJCAI: International Joint Conference on Artificial Intelligence*, pages 720–726, 2016.

11 Eugene C Freuder. Progress towards the holy grail. *Constraints*, 23(2):158–171, 2018.

12 Eugene C Freuder. Conversational modeling for constraint satisfaction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 22592–22597, 2024.

13 Eugene C Freuder and Barry O'Sullivan. Grand challenges for constraint programming. *Constraints*, 19(2):150–162, 2014.

14 Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. E ssence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

15 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

16 Hamza Islah and Younes Mechqrane. Machine learning based recommendation queries for constraint acquisition. In *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 159–165. IEEE, 2024.

17 Serdar Kadıoğlu, Parag Pravin Dakle, Karthik Uppuluri, Regina Politi, Preethi Raghavan, SaiKrishna Rallabandi, and Ravisutha Srinivasamurthy. Ner4opt: named entity recognition for optimization modelling from natural language. *Constraints*, 29(3):261–299, 2024.

18 Mohit Kumar, Samuel Kolb, and Tias Guns. Learning constraint programming models from data using generate-and-aggregate. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

19 Nadjib Lazaar. Parallel constraint acquisition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3860–3867, 2021.

20 Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9), January 2023. doi:10.1145/3560815.

21 Younes Mechqrane, Christian Bessiere, and Ismail Elabbassi. Using large language models to improve query-based constraint acquisition. In *IJCAI 2024-33rd International Joint Conference on Artificial Intelligence*, pages 1916–1925, 2024.

22 Kostis Michailidis, Dimos Tsouros, and Tias Guns. Constraint modelling with llms using in-context learning. In *30th International conference on principles and practice of constraint programming*, 2024.

23 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

24 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

25 Laurent Perron, Frédéric Didier, and Steven Gay. The cp-sat-lp solver (invited talk). In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

26 Kostyantyn Shchekotykhin and Gerhard Friedrich. Argumentation based constraint acquisition. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 476–482. IEEE, 2009.

27 Dimos Tsouros, Senne Berden, and Tias Guns. Generalizing constraint models in constraint acquisition. In *Thirty-Ninth AAAI Conference on Artificial Intelligence, AAAI 2024*, 2025.

28 Dimos Tsouros and Tias Guns. A cpmpy-based python library for constraint acquisition - pycona. In *Proc. AAAI 2025 Bridge on Constraint Programming and Machine Learning (CPML)*, 2025.

29 Dimos Tsouros, Hélène Verhaeghe, Serdar Kadıoğlu, and Tias Guns. Holy grail 2.0: From natural language to constraint models. *arXiv preprint arXiv:2308.01589*, 2023.

30 Dimosthenis C. Tsouros, Senne Berden, and Tias Guns. Learning to learn in interactive constraint acquisition. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, February 20-27, 2024, Vancouver, Canada*, pages 8154–8162. AAAI Press, 2024. URL: https://doi.org/10.1609/aaai.v38i8.28655, doi:10.1609/AAAI.V38I8.28655.

31 Dimosthenis C Tsouros and Kostas Stergiou. Efficient multiple constraint acquisition. *Constraints*, 25(3):180–225, 2020.

32 Dimosthenis C Tsouros and Kostas Stergiou. Learning max-csps via active constraint acquisition. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

33 Dimosthenis C Tsouros, Kostas Stergiou, and Christian Bessiere. Structure-driven multiple constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 709–725. Springer, 2019.

A Feature representation

Our feature representation of constraints $\phi(c)$ is divided into three categories, so that CSs can be extracted using the related features, as was done in [27]:

- *Relation features.* Capture properties of the relation of the given constraint.
- *Partitioning features.* Capture whether the variables in the scope of constraint c have characteristics in common, i.e. are in the same row, column etc.
- *Conditioning features.* Features describing how the variables in the scope of the constraint relate in different ways, e.g. what the distance is between them.

An overview of the feature representation we used in our implementation is given in Table 3.

Table 3 Feature representation of constraints

ID	Name
Relation features	
1	Relation (one-hot-encoded)
2	Has_constant
3	Constant value(s)
Partitioning features	
4_i	dim_i_same
5_{ij}	dim_i_ldim_j_same
Conditioning features	
6_i	dim_i_avg_dist
7_i	dim_i_avg
8_i	dim_i_max
9_i	dim_i_min
10_{ij}	dim_i_ldim_j_avg_dist
11_{ij}	dim_i_ldim_j_abs_avg_dist
12_{ij}	dim_i_ldim_j_avg
13_{ij}	dim_i_ldim_j_max
14_{ij}	dim_i_ldim_j_min

We used 3 relation features, to capture the name of the constraint relation and its constant values. As candidate attributes for partitioning, we used the *indices* of the variables in the dimensions of the tensor they were given in, along with *latent dimensions* that may be discovered using the divisors of the size of each dimension. We include one partitioning feature for each (latent) dimension, expressing whether all variables in the constraint's scope share the same index in them, as in [27]. As additional conditioning features, we used the numerical features derived from the indices, as in [32], i.e., the average, max and min values of the variable indices in each (latent) dimension, along with the average difference and absolute average difference between these indices.

B Additional details for Q2

In this experiment, we focused on the Exam and Nurse benchmarks, which allow controlled instance generation through their parameters, while being the benchmarks that most closely

resemble real-world problems. We generated 10 instances of varying sizes for each, gradually increasing the number of variables and constraints. The exact instances used are shown in Table 4 for Exam, in Table 5 for Nurse, and in Table 6.

Table 4 Exam Timetable instance configurations.

ID	Semesters	Courses/Sem	Slots/Day	Days
1	4	3	4	4
2	4	4	4	5
3	5	4	4	6
4	5	5	4	7
5	6	5	4	8
6	6	6	5	8
7	7	6	5	9
8	7	7	5	11
9	8	7	5	12
10	8	8	5	14

Table 5 Nurse Rostering instance configurations.

ID	Shifts/Day	Days	Nurses	Nurses/Shift
1	3	3	8	2
2	3	4	11	3
3	3	5	11	3
4	3	5	14	4
5	3	6	14	4
6	3	8	15	4
7	3	9	16	4
8	3	10	16	4
9	3	10	16	5
10	3	10	20	6

The scalability results regarding the generalization queries, are shown in Table 7.

C User study details

To evaluate the interpretability of CS-based generalization queries, we collected all such queries that were asked in the experiment of Q1 in 1 run for each benchmark, and we conducted a small-scale user study.

C.1 Translating to natural language using an LLM

Each query was automatically translated into natural language using an LLM. We used gpt-4.1 for this task. We designed a prompt to describe what a CS represents and how to translate them to queries. The input to the LLM also included the variables of the problem, along with their structure and names, as given by the user to the CA system. The generic prompt, describing the CSs and how to translate them, is given in Figure 2. The generated natural language queries for all benchmarks are given in Figures 3-6.

Table 6 Distance-based grid coloring instance configurations.

ID	Rows	Cols	# of colours	Max distance
1	5	5	13	2
2	6	6	13	2
3	7	7	13	2
4	8	7	14	2
5	8	8	14	2
6	9	8	14	2
7	10	8	18	2
8	10	9	18	2
9	10	10	18	2
10	11	11	18	2

Table 7 The effect of the problem size on the number of generalization queries.

	Generalization Queries, Exam Timetabling									
Nr. Constraints:	*77*	*143*	*218*	*348*	*488*	*716*	*959*	*1317*	*1699*	*2231*
QuAcq + GenAcq	20	27	32	42	45	60	63	80	84	97
QuAcq + GQ-Gen	3	4	3	3	4	4	3	6	3	3
	Generalization Queries, Nurse Rostering									
Nr. Constraints:	*53*	*171*	*216*	*394*	*476*	*640*	*722*	*804*	*1275*	*1854*
QuAcq + GenAcq	20	28	32	37	43	50	55	60	66	72
QuAcq + GQ-Gen	5	5	5	7	16	5	5	12	14	10
	Generalization Queries, Grid Coloring									
Nr. Constraints:	*168*	*270*	*396*	*465*	*546*	*627*	*708*	*813*	*918*	*1140*
QuAcq + GenAcq	140	243	360	442	514	580	659	752	836	1316
QuAcq + GQ-Gen	4	6	5	6	9	5	8	5	6	5

For each benchmark, we also gave the description of the benchmark and its variables to the LLM, which are assumed to be known by the user, but without including the descriptions of the requirements/constraints of the problem. Then, the CS-based generalization queries asked in 1 run of each benchmark were given to the LLM one by one, and the natural language queries were added to the study.

C.2 Evaluation

We made an open call to researchers in our department to participate in our user evaluation, and we collected answers from 19 users. For each query, they were asked to:

- Indicate how clear the query was (on a scale from 1 = very unclear to 5 = very clear),
- Provide a yes/no answer to the query (i.e., whether they believed it was valid for the given problem),
- Rate their confidence in their answer (1 = not confident, 3 = very confident),
- Indicate the time they spent on the query (<30s, 30s–1min, >1min).

You are an expert AI assistant specializing in constraint programming and natural language generation. Your task is to translate a structured ‘Constraint Specification‘ into a clear, human-readable question, making sure the question uses the terminology from the user-provided problem description.
The Task
You will be given **two** inputs:
1. A ‘Problem Description‘: This explains the context of the problem, including what each dimension of the variables represents (e.g., days, products etc).
2. A ‘Constraint Specification‘: A formal rule with three components: ‘relation(s)‘, ‘partitioning‘, and ‘sequence conditions‘.
Your goal is to transform the formal ‘Constraint Specification‘ into a single, natural language question that is grounded in the ‘Problem Description‘.
Process to Follow
Understand the Domain
Carefully read the ‘Problem Description‘ to identify the mapping between abstract dimensions (‘Dimension 0‘, ‘Dimension 1‘, etc.) and the real-world concepts they represent. This is the key to creating a domain-specific query.
Translate the Specification Components
Use the understanding from Step 1 to translate each part of the specification.
1. ‘relation(s)‘:
For example: * ‘[(AV4) == (AV5)]‘: This specifies the core relationship. Always translate this to the phrase "...should these variables have equal values?". The specific variable names (e.g., AV4, AV5) are abstract placeholders and should be ignored.
2. ‘partitioning‘:
This defines the partitioning of variables to find the scope over which the rule applies. This part usually forms the beginning of the sentence. Some examples:
* ‘[]‘ (empty list): The rule applies to the entire set of variables.
* ‘[Partition on dimension 0]‘: The rule applies to variables within each **row**.
* ‘[Partition on dimension 1]‘: The rule applies to variables within each **column**.
3. ‘sequence conditions‘:
These are filters that specify which pairs of variables within the defined partitions the rule applies to. Combine these conditions into a descriptive clause. Some examples:
* ‘[]‘ (empty list): The rule applies to **all possible combinations** of variables within the partition. Use a phrase like "should every pair of variables...".
* ‘not_same_dim_0‘: The variables must not be in the same row.
* ‘not_same_dim_1‘: The variables must not be in the same column.
* ‘dim_0_diff‘ (row difference): * ‘operator <=, value 1.5‘: The variables are in the same row or adjacent rows ("at most one row apart"). Notice that in dimension 0, dim_0_diff is always positive, as always the first variable of a relation has a lower value in this dimension than the rest. Take this into account.
* ‘dim_1_diff‘ (column difference): * ‘operator >=, value 0.5‘ AND ‘operator <=, value 1.5‘: The variables are in **adjacent columns**. * ‘operator <=, value -1.5‘ or ‘operator >=, value 1.5‘: The variables are **at least two columns apart**.
* ‘dim_0_max‘ (maximum row index): * ‘operator <=, value 2.5‘: The variables must be located **within the first three rows** (rows 0, 1, and 2). ...

Figure 2 LLM System prompt for translating CS to natural language queries

Exam Timetable Queries

- In the entire timetable, should every pair of exams be assigned to different slots? Answer: True
- For all courses of the first two semesters, should every pair of exams be scheduled on different days? Answer: False
- Within each semester, should every pair of exams be scheduled on different days? Answer: True

Figure 3 Natural language queries used in the user study, from Exam Timetabling

Nurse Rostering Queries

- In the full schedule, should every pair of nurse assignments on the last day be assigned to different nurses? Answer: True

  Within each of the first six days, should every pair of nurse assignments be assigned to different nurses? Answer: True
- In the full schedule, should every pair of assignments for nurse 4 on different days be assigned to different nurses? Answer: False
- In the full schedule, should every pair of nurse assignments where one is on the evening shift of a day and the other is on the morning shift of the following day be assigned to different nurses? Answer: True

Figure 4 Natural language queries used in the user study, from Nurse Rostering

Sudoku Queries

- In the last 3-row block of the grid, should every pair of cells have different values? Answer: False
- Among cells in columns 0 to 3, should every pair of cells have different values? Answer: False
- In the last row of the grid, should every pair of cells have different values? Answer: True
- Among cells in rows 0 to 7 and columns 6 to 8, should every pair of cells have different values? Answer: False
- In each column from 0 to 5, should every pair of cells in rows 0 to 7 have different values? Answer: True
- In each row of the grid, should every pair of adjacent cells have different values? Answer: True
- In each row of the grid, should every pair of cells that are at least two columns apart have different values? Answer: True
- Among cells in columns 0 to 2, should every pair of cells have different values? Answer: False
- Among cells in the same 3-column block and in rows 6 to 8, should every pair of cells in the same or adjacent columns have different values? Answer: False
- Among cells in the same 3x3 block and in rows 0 to 5, should every pair of cells have different values? Answer: True
- In each 3x3 block of the grid, should every pair of cells in the same or adjacent columns have different values? Answer: True

Figure 5 Natural language queries used in the user study, from Sudoku

Greater-Than Sudoku Queries

- In the full grid, should every pair of cells have different values? Answer: False
- In the last two rows of the grid, should every pair of cells have different values? Answer: False
- In each row of the grid, should every pair of cells have different values? Answer: True
- In each column of the grid, should every pair of cells have different values? Answer: True
- In each 3-row block, should every pair of cells in the same row have different values? Answer: False
- Among cells in columns 0 to 5, should every pair of cells in different 3-column blocks have different values? Answer: False
- In each 3x3 block of the grid, should every pair of cells have different values? Answer: True
- Among cells in rows 0 to 6 and columns 0 to 5, should every pair of cells have different values? Answer: False
- Among cells in the top four rows and columns 7 or 8, should every pair of cells have the value in the first cell greater than or equal to the second? Answer: False
- In each 3-column block, should every pair of cells in the same 3-row block have different values? Answer: True
- Among cells in rows 1 or 2 and columns 7 or 8, should every pair of cells have the value in the first cell greater than or equal to the second? Answer: False
- Among cells in row 1 and columns 7 or 8, should every pair of cells have the value in the first cell greater than or equal to the second? Answer: False
- Among cells in column 8, should every pair of cells at most two rows apart have the value in the upper cell less than the lower cell? Answer: False
- In column 8, should every pair of cells in consecutive rows have the value in the upper cell less than the lower cell? Answer: False
- Among cells in column 8 and the same 3-row block, should every pair of cells have the value in the upper cell less than the lower cell? Answer: False
- Among cells in column 8 and the same 3-row block, should every pair of cells have the value in the upper cell less than or equal to the lower cell? Answer: False

Figure 6 Natural language queries used in the user study, from GTSudoku