Generalizing Constraint Models in Constraint Acquisition

Dimos Tsouros[1], Senne Berden[1], Steven Prestwich[2] Tias Guns[1]

[1]Department of Computer Science, KU Leuven, Belgium
[2]School of Computer Science and Information Technology, University College Cork
dimos.tsouros@kuleuven.be, senne.berden@kuleuven.be, s.prestwich@cs.ucc.ie, tias.guns@kuleuven.be

Abstract

Constraint Acquisition (CA) aims to widen the use of constraint programming by assisting users in the modeling process. However, most CA methods suffer from a significant drawback: they learn a single set of individual constraints for a specific problem instance, but cannot generalize these constraints to the parameterized constraint specifications of the problem. In this paper, we address this limitation by proposing GENCON, a novel approach to learn parameterized constraint models capable of modeling varying instances of the same problem. To achieve this generalization, we make use of statistical learning techniques at the level of individual constraints. Specifically, we propose to train a classifier to predict, for any possible constraint and parameterization, whether the constraint belongs to the problem. We then show how, for some classes of classifiers, we can extract decision rules to construct interpretable constraint specifications. This enables the generation of ground constraints for any parameter instantiation. Additionally, we present a generate-and-test approach that can be used with any classifier, to generate the ground constraints on the fly. Our empirical results demonstrate that our approach achieves high accuracy and is robust to noise in the input instances.

Code — https://github.com/Dimosts/GenConModels
Extended Version — https://arxiv.org/abs/2412.14950

Introduction

Constraint Programming (CP) is considered one of the main paradigms for solving combinatorial problems in AI. It provides powerful modeling languages and solvers for decision-making, with many successful applications (Wallace 1996; Simonis 1999). In CP, the user declaratively states the constraints over a set of decision variables, thereby defining the feasible solutions to their problem. A solver is then used to generate a solution. However, modeling a new application as a constraint problem requires significant expertise, which is a barrier to the wider use of CP (Freuder and O'Sullivan 2014; Freuder 2018). This has motivated the development of methods to assist the user in the modeling process (De Raedt, Passerini, and Teso 2018; Freuder 2018; Kolb 2016; Lombardi and Milano 2018). This is

the focus of the research area of *Constraint Acquisition (CA)* (Bessiere et al. 2017), which has been identified as an important topic for CP (De Raedt, Passerini, and Teso 2018) and as progress toward the "Holy Grail" of computer science (Freuder 2018).

In CA, constraints are either learned passively from a set of known solutions and (optionally) non-solutions (Beldiceanu and Simonis 2012; Berden et al. 2022; Prestwich et al. 2021) or actively through interaction with a user (Bessiere et al. 2023; Tsouros and Stergiou 2020, 2021). Recent advancements in both passive and active acquisition systems show significant potential (Prestwich et al. 2021; Prestwich and Wilson 2024; Tsouros, Berden, and Guns 2024). For instance, a recent application of interactive CA in a real-world scheduling problem was presented in (Barral et al. 2024).

However, a significant limitation of most (passive and active) CA systems is that they only learn the ground constraints of a specific problem instance (Arcangioli, Bessiere, and Lazaar 2016; Bessiere et al. 2017; Prestwich 2021; Prestwich et al. 2021; Tsouros, Stergiou, and Sarigiannidis 2018), while in practice it is common for the instance at hand to change over time. To accommodate these changes, well-known constraint modeling languages like MiniZinc (Nethercote et al. 2007) and CPMpy (Guns 2019) allow the use of parameters in the constraint model. An assignment of values to the parameters then instantiates the ground constraints for a given instance of the problem. For example, in exam timetabling with the requirement 'exams of courses that are given in the same semester should be scheduled on different days', the actual ground constraints will be instantiated based on the following parameters: which semester each course belongs to, and how many timeslots are available per day. Achieving the same ability to *generalize* the constraint models learned using CA over different instances of the same problem class has been identified as a key challenge for CA (Simonis 2023).

In the literature on CA, there are only a few works that support generalization. The first approach for generalizing constraints was introduced in interactive CA (Bessiere et al. 2014; Daoudi et al. 2015), to enhance the acquisition process within a single instance. Although it targets within-instance generalization, this approach can be used across instances. Another approach, called *extrapolation*, was re-

cently explored (Prestwich 2022). This method requires learning ground constraints for *multiple* instances, which are then extrapolated to a new instance with different parameters. A limitation is that because it is based on genetic programming, it tends to learn contrived, non-interpretable expressions. A third approach is to learn interpretable parameterized forms of constraints *during* the acquisition process (Lallouet et al. 2010; Kumar, Kolb, and Guns 2022), given examples from different instances of the problem, with the corresponding instance parameters also explicitly given. In (Lallouet et al. 2010) an inductive logic programming approach is used to learn rules of the form *condition* ⇒ *constraint*. On the other hand, COUNT-CP (Kumar, Kolb, and Guns 2022) first learns instance-specific constraints, which are then grouped to obtain *first-order constraints* that can generalize to unseen instances.

In the literature, there is no standardized method for representing parameterized constraint specifications. As a result, the different approaches mentioned target different generalizations based on pre-defined partitions of variables. This prevents them from capturing the diverse range of constraint specifications that may exist in CP models. Finally, methods relying on symbolic search can be expected to struggle in the presence of noise in the input ground constraints, which can obscure the underlying patterns they search for.

In this paper, we first formalize the elements of a parameterized constraint specification, capturing the requirements of a constraint problem in a generic and parameterized way. Then, we propose GENCON, a novel approach to learn such constraint specifications from the ground constraints of given instances of the problem. To achieve that, our method leverages the capabilities of statistical machine learning (ML) to learn complex functions from labeled data. In more detail, our contributions are:

- We introduce an approach for generalizing from one or more given problem instances using constraint-level classification, and we present a parameterized feature representation to capture the constraint specifications.

- For classifiers that allow decision rule extraction, we present a method to translate them into interpretable constraint specifications. These can then be used to generate ground constraints for new problem instances.

- To enable the use of our method with classifiers that do not allow rule extraction, we present a generate-and-test approach that can be used with any classifier.

- We conduct a comprehensive experimental evaluation of our methods, using different classifiers across a range of problem classes and instances. We also show that our approach is robust to noise in the given ground constraints.

Background

We now introduce the necessary concepts used in the paper.

Constraint Satisfaction Problems   A *constraint satisfaction problem* (*CSP*) is a triple $P = (V, D, C)$, defining:

- a set of n decision variables $V = \{v_1, v_2, ..., v_n\}$, representing the entities of the problem,

- a set of n domains $D = \{D_{v_1}, D_{v_2}, ..., D_{v_n}\}$, with $D_{v_i} \subset \mathbb{Z}$ being the domain of $v_i \in V$,

- a constraint set $C = \{c_1, ..., c_t\}$. over the variables in V.

A *constraint* c is a pair $(rel(c), var(c))$, where $var(c) \subseteq V$ is the *scope* of the constraint, and $rel(c)$ is a relation over the domains of the variables in $var(c)$, restricting their allowed value assignments. The *arity* of the constraint, denoted as $|var(c)| = \text{arity}(rel(c))$, indicates the number of variables involved. The set of solutions of a constraint set C is denoted by $sol(C)$.

Constraint Acquisition   In Constraint Acquisition (CA), the pair (V, D) is called the *vocabulary* of the problem at hand and is common knowledge shared by the user and the system. Besides the vocabulary, the learner is also given a *language* Γ consisting of a broad range of *fixed arity* constraint relations that may exist in the problem at hand. Using the vocabulary (V, D) and the constraint language Γ, the system generates the *constraint bias* B, which is the set of all expressions that are 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 that is equivalent to the unknown target constraint set C_T.

Machine Learning Classification   ML classification is a supervised learning task that involves learning a function over a given dataset. The dataset, denoted as $\mathbf{E}$, is a collection of N training examples, $\mathbf{E} = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$. Each example is a pair (x_i, y_i), where x_i is a feature vector from the input space X and y_i is the corresponding class label from the output space Y. The feature vector $\mathbf{x}_i$ is composed of m features, $\mathbf{x}_i = (\phi_{i1}, \phi_{i2}, \ldots, \phi_{im})$, with each feature ϕ_{ij} being a (quantifiable) property of example i. In the case of classification, Y is a set of possible class labels. An ML classifier aims to learn a function $f_\theta : X \to Y$, using a set of learnable parameters θ. These parameters are adjusted during training to minimize a loss function $L(f_\theta(x), y)$ measuring the error between the predicted and actual class labels.

Decision Rules   With the rising importance of explainable AI (XAI) and interpretable ML, various approaches focus on extracting decision rules from ML models (Gilpin et al. 2018). These approaches represent the function $f_\theta : X \to Y$ with if-then rules, denoted as a set $R = \{r_1, r_2, ..., r_k\}$. Each decision rule r_i is a pair (Q_i, y_i), with Q_i being a set of conditions and y_i a class label. Each condition $q_{ij} \in Q_i$ is a function $q_{ij} : X \to \{0, 1\}$ that maps an example x to a binary value indicating whether the given example satisfies the condition. For a rule $r_i = (Q_i, y_i)$ to be satisfied, all of its conditions need to be satisfied, i.e., $q_{ij}(x) = 1 \mid \forall q_{ij} \in Q_i$.

Problem Definition

Constraint problems are often not thought of as a single CSP, but as a set of requirements, with the specific instantiation of the ground CSPs depending on the values of some input parameters $\mathcal{P}$. We illustrate this with the following example, which we will also use as a running example.

Example 1. *Consider a simplified exam timetabling problem with s semesters and n courses per semester. The goal is to schedule the courses' exams over d days, each having t timeslots. The parameters of the problem are $\mathcal{P} = \{s, n, d, t\}$. The requirements are that all exams must be scheduled in different timeslots, while exams of courses from the same semester must be on different days. Different values for the parameters will lead to different ground CSPs for each problem instance. The parameters n and s determine the variables V of the problem, while d and t determine their domains D. The parameter values also determine the set of constraints C. The problem contains* different_day *constraints, which are defined over partitions of variables for courses in the same semester, and the allowed assignments depend on the timeslots per day t.*

Definition 1. *A parameterized constraint problem consists of a set of parameters $\mathcal{P} = \{p_1, p_2, \ldots, p_q\}$, and a function mapping each parameter instantiation $\mathcal{P}_A = \{(p_1, u_{A1}), (p_2, u_{A2}), \ldots, (p_q, u_{Aq})\}$ onto a ground CSP with (V_A, D_A, C_A). The resulting tuple $(\mathcal{P}_A, V_A, D_A, C_A)$ is a* problem instance.

Most CA techniques are aimed at learning the constraint set C_T of a *single* ground CSP, from examples of solutions and non-solutions of that instance. However, in learning, one is often interested in generalizing beyond the instance used for learning, across other instances of the same underlying parameterized constraint problem.

Definition 2. *Given one or more problem instances described by tuples $(\mathcal{P}_A, V_A, D_A, C_A)$, the objective of generalization is to construct a function F such that, for any target problem instance with a vocabulary (V_T, D_T), defined by a unique set of parameter values $\mathcal{P}_T = \{(p_1, u_{T1}), (p_2, u_{T2}), \ldots, (p_q, u_{Tq})\}$, $F(\mathcal{P}_T, V_T, D_T)$ will return the corresponding set of constraints C_T. That is, the aim is to learn the function F from the given ground CSPs, to be able to accurately determine the set of constraints C_T for any target instance with parameters $\mathcal{P}_T$.*

Constraint Specifications

Being able to generalize constraint models involves finding such a function F (definition 2). Typically, in constraint problems, such a function F can be decomposed into several inner functions – which we model as *constraint specifications* (CSs) – each corresponding to a specific requirement of the problem; for example "all courses must be scheduled in a different timeslot". The complete set of constraints C_T of an instance T is then the union of the sets of constraints produced by each inner function.

Each requirement is modeled by a constraint specification, which defines how to derive the pairs $(rel(c), var(c))$ for any target instance T, using the parameter values P_T and the corresponding vocabulary (V_T, D_T). We consider the following three key elements of a CS:

1. Constraint *relation*. The relation $rel(c)$ of each constraint in this CS, which may optionally depend on parameters that determine constant values involved in the relation.

2. Variable *partition(s)*. Typically, a pattern that appears in a constraint model concerns certain *partitions* of variables of the problem and is applied to sequences of variables in this partition. Such partitions can be the dimensions (e.g., rows and columns) of the (multi-dimensional) matrix the variables are given in, or based on latent dimensions in this tensor.

3. *Sequence conditions*. These define which scopes within a partition of variables to apply the constraints to. It is common to have a constraint apply to all possible scopes in a partition. This is done by using the sequence all_pairs for binary constraints, or more generally, the sequence combinations, to find *combinations* of size $arity(r)$ (the arity of the given relation). However, there may also be *sequence conditions*, restricting the variable combinations that should be taken as scopes.

Using these three key elements, we now formally define constraint specifications:

Definition 3. *A constraint specification (CS) is a triple (r, G, S), defining*

- *a relation r, along with any parameters defining its constants,*
- *a variable partition function $G : V_T \to \mathcal{P}(V_T)$, that partitions a given set of variables V_T into subsets based on certain characteristics,*
- *a set of sequence conditions S restricting the scopes to which the constraints are applied.*

A CS can generate the corresponding ground constraints of an instance T using the following generator template:

```
Foreach Y ∈ P(V_T):
  Foreach scope ∈ combinations(Y, arity(r),
                                S):
    c ← (rel(c) = r, var(c) = scope)
```

Example 2. *Consider a problem with the requirement "Values of consecutive variables in the same row must differ". The CS modeling this requirement uses the "$\neq$" relation. The different partitions used in the CS are the rows of the variable matrix. The CS alsone needs to express the sequence condition that the variables need to be consecutive, as it does not apply to all pairs of variables in the same row. In a modeling language, this requirement would be modeled as:*

```
Foreach row ∈ all_rows:
  Foreach v1, v2 ∈ consecutive_pairs(row):
      c ← v1 ≠ v2
```

This is equivalent to our formally defined CS generator, where $\mathcal{P}(V_T)$ corresponds to all_rows *and* consecutive_pairs *corresponds to* combinations *with arity 2, and sequence condition* column(v1) - column(v2) == 1.

Generalizing Constraint Models

To generalize beyond one or more known instances and learn the CSs of a problem, we propose an approach named GEN-CON. The key idea is to use ML to identify patterns in the constraints of the known instance(s) and reconstruct the CSs

of the problem. This is especially promising because, recently, an approach using probabilistic classification during active CA (Tsouros, Berden, and Guns 2024) demonstrated that ML classifiers can effectively detect patterns within the learned constraint network.

GENCON is shown in Figure 1. The given set of ground constraints in the input instance(s) is used in order to *train a classifier* to predict for any constraint whether it belongs to the set of constraints of any target instance of the problem. For this, we use a (parameterized) feature representation of the constraints, whose design is inspired by the different elements of CSs discussed in the previous section. For classes of classifiers allowing the extraction of decision rules, we directly translate these rules to the CSs of the problem, which can produce the ground constraints of any target instance. When decision rules cannot be extracted, a generate-and-test approach is used instead.

Building the Dataset

We build a dataset on the constraint level, i.e., its examples correspond to individual constraints. Given a set of constraints C_A for a problem instance A, and a distinct set of constraints C_A^-, consisting of constraints that are not part of the model, we define a dataset $\mathbf{E}$, wherein each training example is represented as a tuple $(\mathbf{x}_i, y_i)$, corresponding to a constraint $c_i \in C$. For each example $(\mathbf{x}_i, y_i)$, we have $\mathbf{x}_i = \phi_\sigma(c_i, \mathcal{P})$, which is a parameterized feature representation of constraint c_i, and $y_i = [c_i \in C_A]$, a Boolean label that indicates whether c_i is part of the set of true constraints or not.

$$\mathbf{E} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i = \phi_\sigma(c_i, \mathcal{P}) \wedge y_i = [c_i \in C_A], \\ \forall\, c_i \in \{C_A \cup C_A^-\}\}, \quad (1)$$

However, realistically, we may only have access to the set of true constraints C_A for each problem instance. But we also need a set of constraints C_A^- consisting of constraints that will have a negative label, for the classifier to learn how to distinguish between the classes. To produce this set, we first generate a set of constraints B_A, using as a language Γ all relations detected in the given set of constraints C_A, i.e, $\Gamma = \{rel(c) \mid c \in C_A\}$. The bias B_A is created by applying each relation in Γ to all possible scopes in V_A. The set C_A^- then consists of all constraints in B_A that are not part of the given instance(s), i.e., $C_A^- = B_A \setminus C_A$.

Parameterized Feature Representation

In our approach, we propose a framework for the (parameterized) feature representation of constraints, targeted at learning patterns based on the different elements of CSs discussed above. For any constraint c, the classifier expects a fixed-size feature representation $\phi(c)$ as input. As shown in the middle of Figure 1, this feature representation $\phi(c)$ is then transformed to a parameterized version, denoted by $\phi_\sigma(c)$, to be able to learn patterns across instances with different parameter values.

Feature Representation. The feature representation $\phi(c)$ must be designed based on the different elements of CSs that we want the classifier to detect in the problem, i.e., the

relations, partitioning functions, and sequence conditions. Hence, it must contain features that describe the constraint relations, variable characteristics that can be used to recognize partitions of the variables, and other attributes that can play a role in the sequence conditions. Based on this, we construct a feature representation consisting of three groups of features:

1. **Relation features**: Features that capture properties of the relation $rel(c)$ of a given constraint c, along with numerical values of the constants present in the constraint.

2. **Partitioning features**: Features describing whether the variables in the scope of constraint c have characteristics *in common* that can be used in the partitioning function. These characteristics can be problem-specific variable properties (e.g., in what semester a course takes place in exam timetabling), or based on information regarding the structure the variables were given in. For example, in many cases, the variables V are given in the form of a matrix or tensor, and the position of each variable in this tensor often plays a crucial role in the partitioning function of the CSs.

3. **Conditioning features**: These features describe how the variables in the scope of the constraint relate in different ways, to capture sequence conditions that may exist in the CSs of the problem. For example, a constraint may only apply to pairs of variables that are a certain distance away from each other in the variable tensor. Thus, the distance between the variables in a constraint's scope may be included as a conditioning feature. Note that the partitioning features can also be used to capture the sequence conditions, since they describe whether the variables share a certain property or not. For example, a sequence condition may state that the variables must not be part of the same row in the variable matrix.

The grammar of relations, partitioning functions, and sequence conditions used can be considered as the inductive bias of our method. The feature representation needs to be able to capture the CSs existing in the problem at hand. In our implementation, a proof-of-concept feature representation was used based on structural properties of the variables matrix, as matrix modeling is common and beneficial in CP (Flener et al. 2001), with no problem-specific variable attributes.[1] In more detail:

1. We used 3 relation features, describing the name of the constraint relation and its constant values.

2. 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 problem parameters. We include one partitioning feature for each (latent) dimension, expressing whether all variables in the constraint's scope share the same index in them.

3. As additional conditioning features, we used the average difference between the variable indices in each dimension and latent dimension.

---

[1]More information regarding the feature representation used can be found in the appendix (in the extended version).
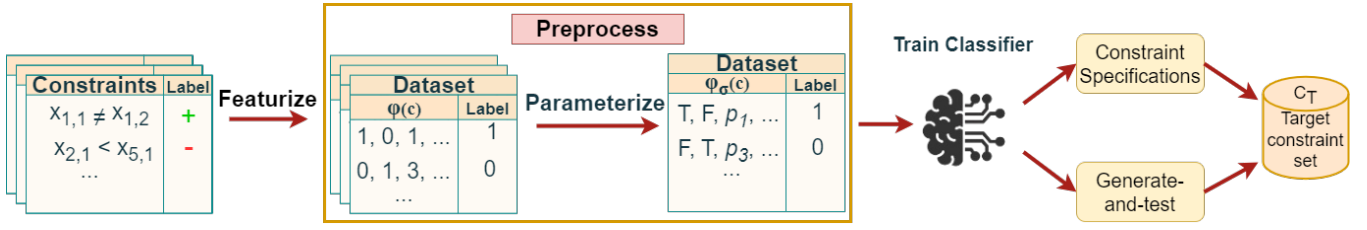
Figure 1: GENCON: Generalizing constraint models through constraint classification, using a parameterized feature representation of constraints

From numerical to categorical features over parameters.
As the goal is to generalize beyond a single problem instance, the feature representation of the constraints should capture the characteristics of the constraints in a generic, parameterized way. Numerical attributes of the constraints typically are not static across instances but depend on parameters of the problem; e.g., in our running example, the constant present in the `different_day` constraints depends on the timeslots-per-day parameter and is not a static value. We want the classifier to be able to capture that. In this step of our approach, we thus replace numerical features with categorical features over the parameters. We do so using a numerical-to-categorical parameter mapping function $\sigma : \mathbb{R} \to \{\text{"NaN"}\} \cup \mathcal{P}_A$ (where P_A is the list of named parameters and their value), defined as follows:

$$\sigma(v) = \begin{cases} p_i & v = u_i \mid (p_i, u_i) \in \mathcal{P}_A \\ \text{"NaN"} & \text{otherwise.} \end{cases} \quad (2)$$

For any numerical feature value that corresponds to a parameter value of the problem instance, function σ replaces the feature by the corresponding parameter's name.

Note that, in a constraint model, it is sometimes not the value u of parameter p that comes up directly in the features. Instead, a trivial arithmetic adaptation of the parameter value may be used, e.g., $u - 1$, $u + 1$ or the multiplication of two parameter values. To capture this, we extend the set of parameters $\mathcal{P}$ with these adaptations, along with the common basic constants 0 and 1, as is also done in COUNT-CP (Kumar, Kolb, and Guns 2022). For the new categorical features, there are thus $|\mathcal{P}| + 1$ categories (where $\mathcal{P}$ is the extended set of parameters): one for every parameter, plus a "NaN" in case none of the parameters match the given value. Our categorical features will thus be able to represent the constraint in a parameterized way. Although arbitrary constants cannot be captured this way, we make the assumption that every constant present is related to the parameters of the problem.

Also note that, when parameterizing the feature representation of a constraint, a single numerical feature value might correspond to multiple parameter values. When this occurs, one example is included in the dataset for each possible matching. Although this could add noise to the dataset, due to examples with a wrong parameter replacing the numerical feature, it ensures that the correct feature representations will definitely be included.

Extracting Constraint Specifications

Decision rules continue to be popular due to their interpretability, with methods existing to extract rules from various classes of classifiers (Barakat and Bradley 2010; Iqbal 2012), in addition to traditional rule-based classification methods. The goal is to derive a set of rules $R = \{r_1, r_2, \ldots, r_k\}$ that represent the classification function. Each rule r_i specifies some conditions Q_i on a subset of features, and a class label y_i, such that $Q_i \Rightarrow y_i$.

In our context, rules that lead to a positive classification define the conditions for a constraint to be part of the target problem. Thus, these conditions can be converted into the CSs of the problem. The extracted CSs can then be used to generate the constraints of any given target instance. We now propose a method for extracting the interpretable CSs of the problem from such a set of learned decision rules. Our approach focuses on the positive-classification rules, iterating over their conditions to identify the relation, partitioning function, and sequence conditions of each CS.

Our method is shown in more detail in Algorithm 1. First, the rules leading to a positive classification are extracted in R_{pos} (line 1). Then, for each rule $r \in R_{pos}$ (line 3), a CS is constructed. The elements of the CS are first initialized (lines 5-7), and then the algorithm iterates over the rule conditions in Q to construct the CS (line 8) as follows:

1. **Relation Extraction**: Identify conditions in Q related to *relation features* (RF) (lines 10-11). These conditions determine which relations from Γ are used, and which constants are used in these relations, if any.

2. **Partitioning Function** (lines 12-14): Identify the partitioning function of the CS using conditions in Q involving *partitioning features* (PF). Use conditions that require certain characteristics to be *equal* in the constraint's variables, and thus can be used to partition the variables based on them.

3. **Sequence Conditions** (lines 15-18): Sequence conditions can be derived from both partitioning features and sequence condition features (SF). More concretely, if a condition Q involves a partitioning feature, and requires certain characteristics to *not* be equal, then this requirement is added to the sequence condition (lines 15-16). If a condition in Q involves a sequence condition feature, it is also added to the sequence conditions (lines 17-18).

Finally, if more than one relation is allowed by the rule's conditions, we create one CS for each, retaining the partitioning function and sequence conditions (lines 19-21).

Algorithm 1: Extracting Constraint Specifications
___

Input: R: a set of decision rules, Γ: a set of relations, RF:
a set of relation features, PF: a set of partitioning fea-
tures, SF: a set of sequence condition features

Output: CS: a set of constraint specifications

1: $R_{pos} \leftarrow \{r \in R \mid y(r) = \text{True}\}$
2: $CS \leftarrow \emptyset$
3: **for all** $r \in R_{pos}$ **do**
4: $\quad Q \leftarrow \text{conditions}(r)$
5: $\quad rel \leftarrow \Gamma$
6: $\quad partition \leftarrow \emptyset$
7: $\quad seq\_cond \leftarrow \emptyset$
8: $\quad$ **for all** $q \in Q$ **do**
9: $\qquad f \leftarrow \text{feature}(q)$
10: $\qquad$ **if** $f \in RF$ **then**
11: $\qquad\quad rel \leftarrow rel \cap \{\gamma \in \Gamma \mid q \text{ is satisfied by } \gamma\}$
12: $\qquad$ **else if** $f \in PF$ **then**
13: $\qquad\quad$ **if** $\text{value}(q) = \text{True}$ **then**
14: $\qquad\qquad partition \leftarrow partition \cup \{q\}$
15: $\qquad\quad$ **else**
16: $\qquad\qquad seq\_cond \leftarrow seq\_cond \cup \{q\}$
17: $\qquad$ **else if** $f \in SF$ **then**
18: $\qquad\quad seq\_cond \leftarrow seq\_cond \cup \{q\}$
19: $\quad$ **for all** $\gamma \in rel$ **do**
20: $\qquad cs \leftarrow \text{create\_CS}(\gamma, partition, seq\_cond)$
21: $\qquad CS \leftarrow CS \cup \{cs\}$
22: **return** CS
___

Example 3. *Recall the exam timetabling problem from Ex-
ample 1, which has two requirements: "all courses must be
scheduled in different timeslots" and "exams of courses from
the same semester must be scheduled on different days".*

*In Figure 2, we can see a decision tree learned to classify
constraints in this problem, using our parameterized feature
representation. Recall that parameter t represents the times-
lots per day. We can extract the following decision rules from
this tree by following the paths from the root to the leaves:*

```
r1: Relation == "different_day"
& Dim0_same == "false"
then 0
r2: Relation == "different_day"
& Dim0_same == "true"
& Constant_parameter != "t"
then 0
r3: Relation == "different_day"
& Dim0_same == "true"
& Constant_parameter == "t"
then 1
r4: Relation == "!=" then 1
```

*Rules $r3, r4$ are the positive-classification rules, which
will be used to construct our CSs. The two CSs that will be
extracted, along with their generators, are:*

1. *CS_1: Relation: "different_day(t)", Partitioning attribute:
dim_0_same, Sequence conditions: $\emptyset$*

```
Foreach row ∈ all_rows:
  Foreach scope ∈ all_pairs(row):
    c ← (rel(c) = "different_day(t)",
                var(c) = scope)
```
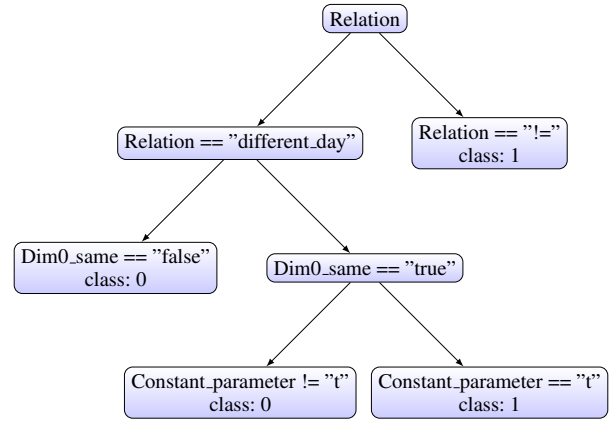

Figure 2: Decision Tree for Exam Timetabling in Example 3

2. *CS_2: Relation: "!=", Partitioning attribute(s): None, Se-
quence conditions: $\emptyset$*

```
Foreach scope ∈ all_pairs(V):
  c ← (rel(c) = "!=", var(c) = scope)
```

Generate-and-Test

To enable the use of our method even when decision rules
cannot be extracted, we now present a generate-and-test ap-
proach that can be used with any classifier, as an alternative
to extracting the CSs from interpretable classifiers.

The intuition of this approach is the following: Even if we
cannot extract the CSs from the learned classifier f_θ explic-
itly, we know that it implicitly represents them. Thus, we can
use the classifier itself to recognize the true constraints for
any problem instance. Our *generate-and-test* approach does
so by generating a set of candidate constraints B_T for the tar-
get problem T, using the language Γ as described above. For
relations with constants, the set $\mathcal{P}$ provides candidate values.
To decide which of the constraints from B_T to use, each of
them is featurized, and the function f_θ predicts whether it
should be part of the model. We keep all constraints with
positive classification:

$$C_T = \{c \mid c \in B_T \land f_\theta(\phi_\sigma(c, \mathcal{P})) = \text{True}\} \qquad (3)$$

Experimental Evaluation

We now experimentally evaluate GENCON, using ground
CSPs of different instances on a variety of benchmarks.
We evaluate our approach both when the given sets of con-
straints are correct and when noise exists. Noisy CSPs can
result when the ground CSPs were themselves acquired us-
ing passive CA, on a noisy dataset of solutions and non-
solutions, or on a dataset containing too few examples. We
recognize two different types of noise in our setting:

1. False positive (FP) noise, where the input set of con-
straints is not sound, also including wrong constraints.

2. False negative (FN) noise, where the input set of con-
straints is not complete, missing some true constraints.

We aim to answer the following experiment questions:

(Q1) To what extent does GENCON effectively generalize ground CSPs?

(Q2) What is GENCON's performance when the input set of constraints also includes wrong constraints?

(Q3) What is GENCON's performance when the input set of constraints does not include all true constraints?

Experimental Setup

Benchmarks. We focused on using benchmarks that have different constraint specifications so that our method is evaluated in distinct cases. Namely, we used the following benchmarks that are commonly used in CA: Sudoku, Golomb, Exam Timetabling (ET) and Nurse Rostering (NR). In each benchmark, we used 10 instances with different parameters.[2] We employed a challenging variant of leave-one-out cross-validation, referred to as *leave-one-in cross-validation*: for each fold, we used just a single instance for training and the remaining nine instances for testing. We present the average results of this process.

Metrics. We evaluate each method by identifying the correctly generated constraints and the number of constraints missing from the model of the target instance(s). Note that, for a constraint to have a positive ground truth label, our evaluation did not only check if a constraint is part of the ground truth model but also if it is logically implied by it. Based on that, we define as True Positives (TP) the correctly identified constraints, as False Positives (FP) the incorrectly identified constraints, and as False Negatives (FN) the missing constraints. Using the defined concepts, our evaluation is based on the following metrics that are common in ML:

- **Precision (Pr)**: It measures the accuracy of the identified constraints in the target instance. A high precision score signifies a low rate of false positives. When the precision score is 100%, the predicted set of constraints is sound.

- **Recall (Re)**: It measures the method's ability to identify all relevant constraints. A high recall score indicates a low rate of false negatives. When the recall score is 100%, the predicted set of constraints is complete.

Comparison. To obtain the CSs of the problem from extracted decision rules, we used Decision trees (DT) and the rule-based classifier CN2. Then, these CSs were used to generate the ground CSPs of the target instances. We also evaluated the generate-and-test approach with a variety of classifiers: Random Forests (RF), Naive Bayes (NB), Multi-layer Perceptron (MLP), and K-Nearest Neighbours (KNN). We used CN2, DT, RF, and NB with their default parameters and tuned the most important hyperparameters for MLP and KNN.[2] We compare our method with the generalization approach used in COUNT-CP (Kumar, Kolb, and Guns 2022).

For the experiments that evaluate the impact of noise, we changed the ground CSPs for the input instances, injecting noisy constraints in them. We evaluated our method on 4 different levels of noise (5%, 10%, 15%, 20%) w.r.t. the original size of the input set of constraints C_A. To inject FP noise, we randomly add the respective percentage of constraints

---

[2]Details can be found in the appendix.
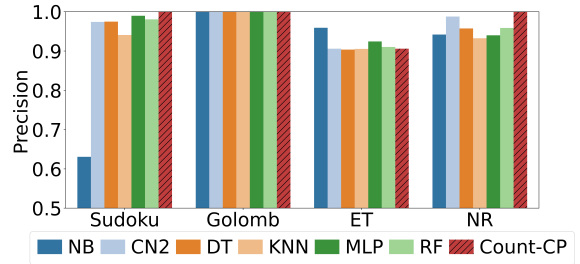
(a) Precision
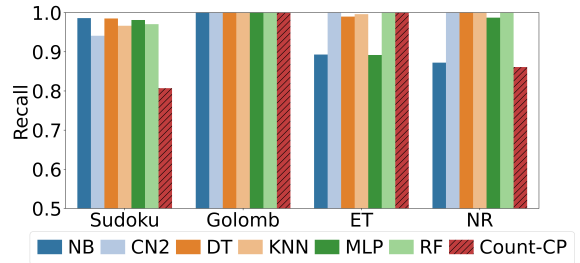
(b) Recall

Figure 3: Results comparing our method (using different classifiers) with COUNT-CP generalization

from the set constraints C_A^- in C_A, while for FN noise, we directly remove constraints randomly from C_A.

Implementation and hardware All experiments were conducted on a system with an Intel(R) Core(TM) i7-2600 CPU, 3.40GHz clock speed, with 16 GB of RAM. All methods and benchmarks were implemented in Python. We used the CPMpy library (Guns 2019) for constraint modeling, and the Scikit-Learn library (Pedregosa et al. 2011) for the ML classifiers, except CN2, for which the Orange library (Demšar et al. 2013) was used. For COUNT-CP, in the available implementation[3] the generalization is mixed with learning, so we re-implemented it stand-alone.

Results

Q1: To what extent does GENCON effectively generalize ground CSPs? Figure 3 shows the results of GENCON using different classifiers, and of COUNT-CP's generalization. Both the extraction of CSs for DT and CN2, as well as generate-and-test for the other classifiers, achieve high precision and recall in all benchmarks. The only exception is NB, which gets lower recall in ET and NR, and significantly lower precision in Sudoku. We believe that this is due to its feature independency assumption, which makes it hard for the classifier to recognise the relationship of the different features in the CSs. The benchmark that turned out to be the most difficult for all methods was ET, with the difficulty being recognizing the parameter of the CS regarding the `different_day` constraints. The problem occurred in instances with many parameters with the same value, with all

---

[3]https://github.com/ML-KULeuven/COUNT-CP

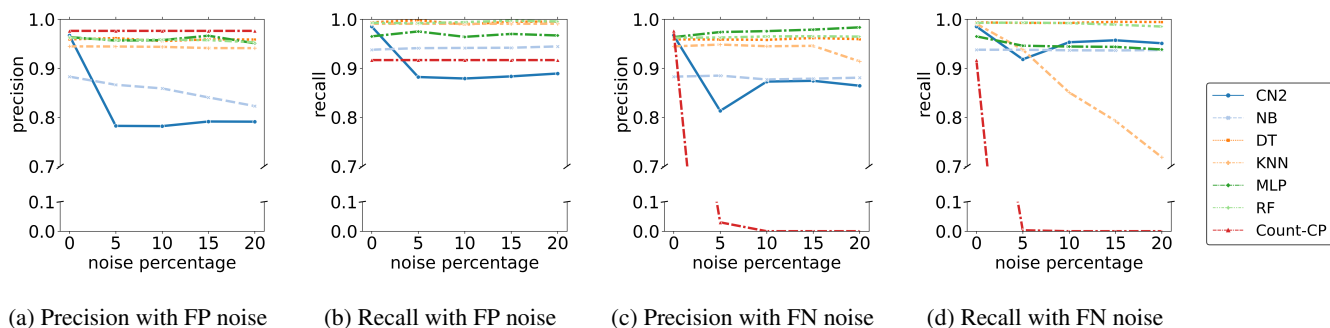|	
(a) Precision with FP noise	(b) Recall with FP noise
(c) Precision with FN noise	(d) Recall with FN noise

Figure 4: (a) and (b) Results with the presence of FP noise in the input constraint model, (c) and (d) Results with the presence of FN noise in the input constraint model (best viewed in color).

of them being recognized as part of the `different_day` CS. This led to the generation of additional constraints in the target instances, lowering the precision.

COUNT-CP also demonstrates good performance in general. Besides ET, where it presents the same issue as our method, it achieves 100% *precision* in the other three benchmarks. However, its main drawback is illustrated in the *recall* results, as it was not able to capture all CSs in NR and Sudoku. In NR, the CSs regarding consecutive shifts cannot be captured, as COUNT-CP does not include sequence conditions in its generalization approach, and only searches for patterns that apply in all sequences of predefined partitions. In Sudoku, the block partitions are not automatically found.

COUNT-CP includes an option to manually give custom partitions as input, and in this case, its recall in Sudoku is increased to 100%. Notably, when we include special features for these custom partitions in our approach, the results with all classifiers also increase to 100%.

Q2: What is the impact of FP noise in the performance of GENCON? Due to space limitations, we present the average results over all benchmarks for each method.[4] The precision results are shown in Figure 4a, while the recall results are shown in Figure 4b.

We can observe that the CN2 classifier (and thus also the CSs extracted from it) and NB are the most sensitive to false positives, presenting increasingly worse precision scores when noise increases. This is because the learning approach of CN2 is not very tolerant to this kind of noise, overfitting in many cases to non-existing patterns. When any of the other classifiers are used in GENCON, their performance remains about as good as in the noiseless setting. The classifiers that already presented high scores in Q1 stay around 95-100%, even when the noise percentage reaches 20%. Similarly, the COUNT-CP generalization approach keeps the same performance as in the original results without noise. That is because it only searches for specific partition patterns and symbolic expression bounds, and thus the randomly inserted constraints are directly disregarded.

Q3: What is the impact of FN noise on the performance of GENCON? As in Q2, we present the average results

[4]Detailed results per benchmark can be found in the appendix.

over all benchmarks. The precision results are shown in Figure 4c, while the recall results are shown in Figure 4d.

We can observe that KNN is the most sensitive to FN noise, with worsening recall when more noise is added, meaning that it struggles to find all the constraints of the target instance. For CN2, the lack of noise tolerance shows up again, as in Q2, though precision and recall stay above 80%. When any of the other classifiers is used, results remain good, even for up to 20% noise. These results demonstrate the ability of our classification-based approach to generalize even in the presence of high percentages of noise.

On the other hand, in the presence of false negatives, the COUNT-CP generalization fails to detect any patterns and does not find any constraints in the target instances, as it searches for partitions in which all sequences of variables share a given constraint. Importantly, COUNT-CP fails to generalize, even when only 5% noise is added.

Conclusions

CP models are typically defined by parameterized specifications, rather than a flat list of ground constraints. However, most CA methods focus on learning a single ground CSP for a specific instance. Our work addresses this limitation by generalizing ground CSPs to parameterized models using a constraint-level classification approach named GENCON. We showed how interpretable CSs can be derived from decision rules, and introduced a generate-and-test method for non-interpretable classifiers. Our evaluation indicates that GENCON achieves high accuracy and robustness, even for high levels of noise, highlighting the potential of ML-based techniques for generalizing constraint models and making CA more robust. We recommend using decision trees as the classifier of choice, as they facilitate the extraction of interpretable CSs while presenting strong performance.

Promising avenues for future work include exploring active learning to enhance generalization; and using generalization during interactive constraint learning to reduce queries, leveraging also the noise robustness demonstrated here. Additionally, GENCON can be applied during passive CA, enabling the learning of constraint models from a limited amount of solutions and non-solutions across various instances, a scenario common in real-world applications.

Acknowledgments

References

Arcangioli, R.; Bessiere, C.; and Lazaar, N. 2016. Multiple Constraint Acquisition. In *25th International Joint Conference on Artificial Intelligence*.

Barakat, N.; and Bradley, A. P. 2010. Rule extraction from support vector machines: a review. *Neurocomputing*, 74(1-3): 178–190.

Barral, H.; Gaha, M.; Dems, A.; Côté, A.; Nguewouo, F.; and Cappart, Q. 2024. Acquiring Constraints for a Nonlinear Transmission Maintenance Scheduling Problem. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer.

Beldiceanu, N.; and Simonis, H. 2012. A model seeker: Extracting global constraint models from positive examples. In *Principles and practice of constraint programming*, 141–157. Springer.

Berden, S.; Kumar, M.; Kolb, S.; and Guns, T. 2022. 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)*.

Bessiere, C.; Carbonnel, C.; Dries, A.; Hebrard, E.; Katsirelos, G.; Narodytska, N.; Quimper, C.-G.; Stergiou, K.; Tsouros, D. C.; and Walsh, T. 2023. Learning constraints through partial queries. *Artificial Intelligence*, 319: 103896.

Bessiere, C.; Coletta, R.; Daoudi, A.; Lazaar, N.; Mechqrane, Y.; and Bouyakhf, E.-H. 2014. Boosting Constraint Acquisition via Generalization Queries. In *ECAI*, 99–104.

Bessiere, C.; Koriche, F.; Lazaar, N.; and O'Sullivan, B. 2017. Constraint acquisition. *Artificial Intelligence*, 244: 315–342.

Daoudi, A.; Lazaar, N.; Mechqrane, Y.; Bessiere, C.; and Bouyakhf, E. H. 2015. Detecting types of variables for generalization in constraint acquisition. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, 413–420. IEEE.

De Raedt, L.; Passerini, A.; and Teso, S. 2018. Learning constraints from examples. In *Proceedings in Thirty-Second AAAI Conference on Artificial Intelligence*.

Demšar, J.; Curk, T.; Erjavec, A.; Gorup, Č.; Hočevar, T.; Milutinovič, M.; Možina, M.; Polajnar, M.; Toplak, M.; Starič, A.; et al. 2013. Orange: data mining toolbox in Python. *the Journal of machine Learning research*, 14(1): 2349–2353.

Flener, P.; Frisch, A.; Hnich, B.; Kiziltan, Z.; Miguel, I.; and Walsh, T. 2001. Matrix modelling. In *Proc. of the CP-01 Workshop on Modelling and Problem Formulation*, 223.

Freuder, E. C. 2018. Progress towards the Holy Grail. *Constraints*, 23(2): 158–171.

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

Gilpin, L. H.; Bau, D.; Yuan, B. Z.; Bajwa, A.; Specter, M.; and Kagal, L. 2018. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, 80–89. IEEE.

Guns, T. 2019. 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.

Iqbal, M. R. A. 2012. Rule extraction from ensemble methods using aggregated decision trees. In *Neural Information Processing: 19th International Conference, ICONIP 2012, Doha, Qatar, November 12-15, 2012, Proceedings, Part II 19*, 599–607. Springer.

Kolb, S. M. 2016. Learning constraints and optimization criteria. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*.

Kumar, M.; Kolb, S.; and Guns, T. 2022. 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.

Lallouet, A.; Lopez, M.; Martin, L.; and Vrain, C. 2010. On Learning Constraint Problems. In *Proceedings of the IEEE International Conference on Tools With Artificial Intelligence*, 45–52.

Lombardi, M.; and Milano, M. 2018. Boosting Combinatorial Problem Modeling with Machine Learning. In Lang, J., ed., *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, 5472–5478.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *International Conference on Principles and Practice of Constraint Programming*.

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

Prestwich, S. 2021. Unsupervised Constraint Acquisition. In *33rd International Conference on Tools with Artificial Intelligence*.

Prestwich, S. 2022. Extrapolating Constraint Networks by Symbolic Classification. In *5th IJCAI Workshop on Data Science Meets Optimization*.

Prestwich, S. D.; Freuder, E. C.; O'Sullivan, B.; and Browne, D. 2021. Classifier-based constraint acquisition. *Annals of Mathematics and Artificial Intelligence*, 1–20.

Prestwich, S. D.; and Wilson, N. 2024. A Statistical Approach to Learning Constraints. *International Journal of Approximate Reasoning, Special Issue on Synergies Between Machine Learning and Reasoning*, 171. To appear.

Simonis, H. 1999. Building industrial applications with constraint programming. In *International Summer School on Constraints in Computational Logics*, 271–309. Springer.

Simonis, H. 2023. Requirements for Practical Constraint Acquisition. In *AAAI 2023 Bridge on Constraint Programming and Machine Learning*.

Tsouros, D. C.; Berden, S.; and Guns, T. 2024. Learning to Learn in Interactive Constraint Acquisition. In Wooldridge, M. J.; Dy, J. G.; and Natarajan, S., eds., *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, February 20-27, 2024, Vancouver, Canada*, 8154–8162. AAAI Press.

Tsouros, D. C.; and Stergiou, K. 2020. Efficient multiple constraint acquisition. *Constraints*, 25(3): 180–225.

Tsouros, D. C.; and Stergiou, K. 2021. 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.

Tsouros, D. C.; Stergiou, K.; and Sarigiannidis, P. G. 2018. Efficient Methods for Constraint Acquisition. In *24th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science vol. 11008*, 373–388.

Wallace, M. 1996. Practical applications of constraint programming. *Constraints*, 1(1-2): 139–168.