Towards a formalization of configuration problems for ASP-based reasoning: Preliminary report

Nicolas Rühling[1,2,*], Torsten Schaub[1,2] and Tobias Stolzmann[1,2]

[1]University of Potsdam, Germany
[2]Potassco Solutions, Germany

Abstract

We develop a principled approach to configuration that targets Answer Set Programming by integrating established concepts in a uniform setting. We begin by defining an abstract specification of configuration problems, drawing on concepts from the literature. We define both, user requirements and configuration solutions, as (partial) instantiations of a configuration model, and require the latter to be an extension of the former. The core of our configuration models comprise a partonomic structure which is adorned with constraints over atomic and aggregated attributes. Driven by this principled approach, we then develop a domain-independent ASP encoding for configuration.

Keywords

Answer Set Programming, Configuration, Encoding

1. Introduction

Configuration has been a central topic in AI since several decades [1, 2, 3]. Early on already, non-monotonic formalisms emerged as a promising alternative for modeling configuration problems [4]. Nowadays, this role is filled by Answer Set Programming (ASP) [5], a non-monotonic problem solving paradigm, combining an easy, rule-based modeling language with high performance solving capacities [6, 7]. Over the years, this has led to several applications of ASP to configuration problems, among them [8, 9, 10] and notably the ASP-based configuration systems WeCoTin [11] and VariSales [12].

Our objective is to develop a principled ASP-oriented approach to configuration, which integrates established concepts from configuration in a uniform setting. However, while ASP usually strives for generality, aiming at problem encodings covering the greatest possible class of problems, many approaches to configuration appear to be more down-to-earth, targeting more specific classes of configuration problems. Hence, as an intermediate step, we begin by defining an abstract specification of configuration problems, drawing on concepts borrowed from [13, 2, 14]. More precisely, we describe configuration problems in terms of a configuration model, user requirements and resulting configuration solutions [15]. Both user requirements and solutions are defined as (partial) instantiations of the configuration model [13], where the latter is required to be an extension of the former. The core of our configuration models comprise a partonomic structure which is adorned with constraints over atomic and aggregated attributes.

Driven by this principled approach, we then develop a domain-independent ASP encoding for configuration.

The paper is structured as follows. In section 2 we formally define a configuration problem and its solutions. Section 3 discusses how constraints and aggregation of values are handled. In section 4 we present the ASP fact format and encoding and show how to solve configuration problems using the ASP solver *clingo*. Section 5 gives an overview of related work and section 6 concludes the paper.

2. Configuration problem and solutions

We represent configuration problems as (configuration) models along with one of their (partial) instantiations. Formally, both are expressed in terms of (directed) multigraphs;[1] the model's graph delineates the ones capturing partial instantiations. User requirements and solutions are both represented by instantiations.

A *configuration problem* is a pair (M, I). A simple example is given in Figure 1. The (configuration) model M is a tuple $(T, P, s_p, t_p, D, V, E, C, de, at, co)$, where

1. (T, P, s_p, t_p) is a multigraph, where
   a) T is a set of *types*,
   b) $P = P_P \cup P_C$ is a partition of *ports*, where
      i. P_P is a set of *partonomic ports*,
      ii. P_C is a set of *connection ports*,

[1]A *multigraph* is a graph admitting more than one arc between two vertices; for this, we use edges with own identity.

ConfWS'23: 25th International Workshop on Configuration, Sep 6–7, 2023, Málaga, Spain
*Corresponding author.
nruehling@uni-potsdam.de (N. Rühling)
0000-0001-5157-6788 (N. Rühling); 0000-0002-7456-041X (T. Schaub); 0000-0002-1436-0715 (T. Stolzmann)

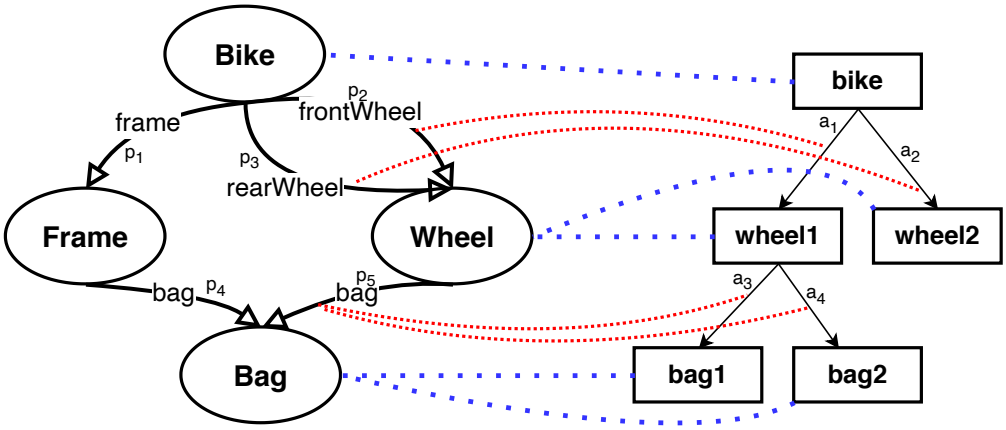

Figure 1: Example of a simple configuration problem.

   c) $s_p : P \to T$ assigns a port its *source type*,
   d) $t_p : P \to T$ assigns a port its *target type*, and
   e) (T, P_P, s_p, t_p) is a rooted acyclic graph,
2. $D = D_P \cup D_A$ is a partition of *descriptors*, where
   a) D_P is a set of *port descriptors*,
   b) D_A is a set of *attribute descriptors*,
3. V is a set of *values*,
4. E is a set of *evaluators*,
5. C is a set of *table constraints*,
6. $de : P \to D_P$ assigns a port its port descriptor, such that
   if $s_p(p) = s_p(p')$ and $de(p) = de(p')$ then $p = p'$ for all $p, p' \in P$,
7. $at : T \to 2^{D_A \times E}$ assigns a type its set of attribute descriptors and evaluators, such that
   for any type $t \in T$, if $(d, e), (d, e') \in at(t)$ then $e = e'$, and
8. $co : T \to 2^C$ assigns a type its set of constraints.

We often refer to (T, P, s_p, t_p) as the *model graph*, and to (T, P_P, s_p, t_p) as the *partonomy* (graph); its root represents the configured object.

An example of a model graph is given on the left in Figure 1. It consists of four types, $T = \{Bike, Frame, Wheel, Bag\}$, linked by five partonomic ports, viz. $P_P = \{p_1, p_2, p_3, p_4, p_5\}$ with source $s_p(p_1) = s_p(p_2) = s_p(p_3) = Bike$, $s_p(p_4) = Frame$, $s_p(p_5) = Wheel$ and target $t_p(p_1) = Frame$, $t_p(p_2) = t_p(p_3) = Wheel$ and $t_p(p_4) = t_p(p_5) = Bag$. The corresponding descriptors are $de(p_1) = frame$, $de(p_2) = frontWheel$, $de(p_3) = rearWheel$ and $de(p_4) = de(p_5) = bag$. Note that two ports can have the same descriptor as long as their source type is different. In usual graph terminology, this amounts to two edges $(Bike, Wheel)$ labeled with $frontWheel$ and $rearWheel$, respectively. A third edge $(Bike, Frame)$ labeled with $frame$ and a fourth and fifth edge $(Frame, Bag)$, resp. $(Wheel, Bag)$, both labeled with bag. The other components of a configuration model are detailed in Section 3.

An instantiation I of M is a tuple $(O, A, s_a, t_a, m_O, m_A, X, v)$, where

1. (O, A, s_a, t_a) is a multigraph, where
   a) O is a set of *objects*,
   b) A is a set of *associations*,
   c) $s_a : A \to O$ assigns an association its *source object*,
   d) $t_a : A \to O$ assigns an association its *target object*,
2. $m_O : O \to T$ maps objects to types,
   $m_A : A \to P$ maps associations to ports, such that for all $a \in A$
   a) $m_O(s_a(a)) = s_p(m_A(a))$, and
   b) $m_O(t_a(a)) = t_p(m_A(a))$,
3. $X = \{(o, d) \mid o \in O, (d, e) \in at(m_O(o))$ for some $e \in E\}$ is a set of *attribute variables*, and
4. $v : X \to V$ maps attribute variables to values.

For simplicity, we sometimes drop the subscripts of m_O and m_A and simply write m, when clear from the type of argument.

An example instantiation of the configuration model in Figure 1 is given on its right. It includes objects $O = \{bike, wheel1, wheel2, bag1, bag2\}$ whose relationships are fixed via the associations $A = \{a_1, a_2, a_3, a_4\}$ with source $s_a(a_1) = s_a(a_2) = bike$ and $s_a(a_3) = s_a(a_4) = wheel1$, and target $t_a(a_1) = wheel1$, $t_a(a_2) = wheel2$, $t_a(a_3) = bag1$, and $t_a(a_4) = bag2$. The actual instantiation of the configuration model is warranted by functions m_O and m_A. The object mappings are

$m_O(bike) = Bike$, $m_O(wheel1) = m_O(wheel2) = Wheel$, and $m_O(bag1) = m_O(bag2) = Bag$. The association mappings are $m_A(a_1) = p_2$ and $m_A(a_2) = p_3$, and $m_A(a_3) = m_A(a_4) = p_5$. There are no corresponding objects and associations for type *Frame* and partonomic port *frame*, respectively. This shows that partial instantiations are fully admissible.

The other components of instantiations are detailed in Section 3.

Finally, a *valid* instantiation I of M satisfies the following conditions:

1. All constraints in $co(m(o))$ are satisfied for all $o \in O$, and
2. the subgraph (O, A_P, s_a, t_a), where $A_P = \{a \in A \mid m_A(a) \in P_P\}$ is the set of all *partonomic associations*, is a tree with root $r \in O$ such that $m_O(r)$ is the (partonomic) root of (T, P_P, s_p, t_p).

The satisfaction of constraints is detailed in Section 3. The first condition ensures consistency of the instantiation while the second guarantees that it is indeed a configuration of the object in focus where every non-root object is a part of exactly one other object.

For comparing instantiations in terms of partiality, we view all components as sets (i.e., functions as relations) and compare them with set inclusion. Accordingly, we say that an instantiation I' is an *extension* of another I, written $I \prec I'$, if all components of I are subsets of the ones of I'. In this way, we may pose a configuration problem (M, U) as a configuration model M along with *user requirements* U expressed as a (partial) instantiation of M. We define the set of *solutions* to (M, U) as

$$S(M, U) = \{I \mid I \text{ is a valid instantiation of } M \text{ and } U \preceq I\}.$$

This assures the user requirements are included in any solution; invalid user requirements or ones which cannot be extended to a valid instantiation may lack solutions.

A *minimal* solution of some user requirements U is a valid extension $U \prec I$ such that there is no valid instantiation I' with $U \prec I' \prec I$.

One might wonder why a rigourous specification of configuration problems as shown above is necessary. Such a specification allows us to show properties of our formalism. For example, there is a simple proof that the solution space behaves monotonically for a fixed model.

Proposition 1. *Let M be a fixed configuration model. Then for any user requirements U and U' it holds that $U \prec U'$ implies $S(M, U') \subseteq S(M, U)$.*

Proof. Take any $I \in S(M, U')$. Per definition I is valid and $U' \prec I$, that is, all components of U' are subsets of I. We also have $U \prec U'$ so all components of U are subsets of U'. That means all components of U are also subsets of I and thus $U \prec I$. It follows that $I \in S(M, U)$. □

Note that in general this does not hold for minimal solutions.

3. Constraint handling and aggregation

The objects in a valid instantiation must satisfy all associated constraints in the underlying configuration model. Apart from the structural constraints imposed by the model graph, additional constraints can be imposed on objects (in the instantiation) via their types.

As an example, consider Figure 2 showing the model graph from Figure 1 but with attributes and constraints. The model is extended by a set of constraints $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, assigned to their corresponding types, viz. $co(Bike) = \{c_1, c_2, c_3\}$, $co(Wheel) = \{c_4, c_5\}$, and $co(Bag) = \{c_6\}$.

The attributes are assigned to their types as follows

$$\begin{aligned} at(Bike) &= \{(maxWeight, \top), (minStowage, \top), \\ &\qquad (totalWeight, e_1), (stowage, e_2)\}, \\ at(Wheel) &= \{(size, \top), (weight, \top)\}, \text{ and} \\ at(Bag) &= \{(volume, \top), (weight, \top)\}. \end{aligned}$$

Here $\top, e_1, e_2 \in E$ are evaluators whose function is explained later in this section. While constraint c_1 guarantees that the front and rear wheel of a bicycle have the same size, constraints c_2 and c_3 assure that the values of the total weight and stowage of the bike lie within some (possibly user-requested) range. Constraints c_4 and c_5 specify possible combinations of the attributes of the wheels and bags. Lastly, constraint c_6 expresses that only small bags can be attached to a wheel.

We represent constraints in their canonical form as tables. For illustration of how table constraints work, consider the instantiation in Figure 3 and constraint $c_1 \in co(m(bike))$ from Figure 2. This constraint is expressed as an equality but can easily be rewritten as a table containing all combinations which satisfy the given relation. The constraint describes compatible values of the attribute *size* of *Wheel* at paths *frontWheel* and *rearWheel* of *Bike*.

To make this relation precise, we rely on *path expressions* leading from the type at hand to the attributes in focus. In our example, they are given in the header of the table constraint. Notably, given that this structure is mirrored in corresponding instantiations, the path expressions also allow us to access the values of these attributes from each object of the type at hand.

More precisely, a *path expression* is a finite sequence of descriptors. We distinguish path expressions only including port descriptors in D_P, and the ones consisting

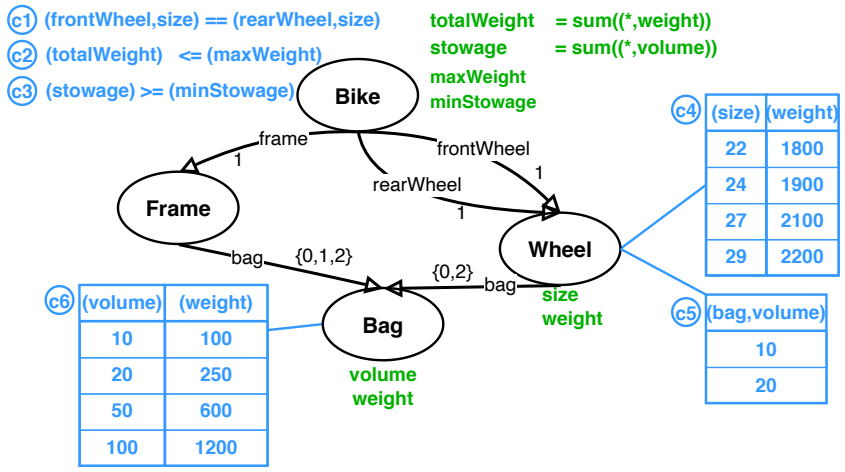

Figure 2: Model of bike example with constraints.

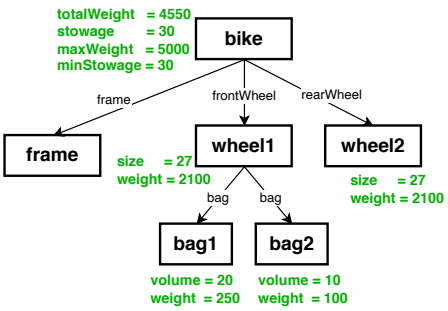

Figure 3: Instantiation of bike example.

of a sequence of port descriptors followed by a single attribute descriptor from D_A. We refer to them as port and attribute path expressions, respectively. Attribute path expressions are used as attributes in table constraints, as with $(frontWheel, size)$ or $(weight)$ in Figure 2.

With this, a *table constraint* is a pair $((\vec{d_1}, \ldots, \vec{d_n}), R)$ where each $\vec{d_i}$ is an attribute path expression for $1 \leq i \leq n$ and $R \subseteq V^n$ is an n-ary relation over V. We use path expressions to select objects as well as attribute variables. For an object $o \in O$ and $n \geq 0$, we define

$$sel_o(\epsilon) = \{o\} \text{ for the empty path sequence } \epsilon. \quad (1)$$

$$\begin{aligned} &sel_o((d_1, \ldots, d_n, d_{n+1})) = \\ &\quad \{o' \in O \mid a \in A, t_a(a) = o', \\ &\quad s_a(a) \in sel_o((d_1, \ldots, d_n)), de(m(a)) = d_{n+1}\} \\ &\quad \text{if } d_{n+1} \in D_P \text{ and } d_i \in D_P \text{ for } 1 \leq i \leq n. \end{aligned} \quad (2)$$

$$\begin{aligned} &sel_o((d_1, \ldots, d_n, d_{n+1})) = \\ &\quad \{(o', d_{n+1}) \in X \mid o' \in sel_o((d_1, \ldots, d_n))\} \\ &\quad \text{if } d_{n+1} \in D_A \text{ and } d_i \in D_P \text{ for } 1 \leq i \leq n. \end{aligned} \quad (3)$$

In our example, for constraint $c_1 \in co(m(bike))$ we get

$$sel_{bike}((frontWheel)) = \{wheel1\} \quad (4)$$

$$sel_{bike}((rearWheel)) = \{wheel2\} \quad (5)$$

$$sel_{bike}((frontWheel, size)) = \{(wheel1, size)\} \quad (6)$$

$$sel_{bike}((rearWheel, size)) = \{(wheel2, size)\}. \quad (7)$$

While (6) and (7) give attribute path expressions, (4) and (5) give port path expressions.

Given an object $o \in O$ and a valuation v, o satisfies a table constraint $((\vec{d_1}, \ldots, \vec{d_n}), R)$ in $co(m(o))$, if $(v(x_1), \ldots, v(x_n)) \in R$ for every $(x_1, \ldots, x_n) \in sel_o(\vec{d_1}) \times \cdots \times sel_o(\vec{d_n})$.

For the type *bike* in our example in Figure 3, we continue the illustration of constraint $c_1 \in co(m(bike))$ given by

$$\begin{aligned} &(((frontWheel, size), (rearWheel, size)), \\ &\{(22, 22), (24, 24), (27, 27), (29, 29)\}) \end{aligned}$$

When using the two attribute path expressions to select attribute variables, we look at the cross product of (6) and (7):

$$\{((wheel1, size), (wheel2, size))\} \quad (8)$$

Applying the valuation v from Figure 3, namely,

$$v = \{(wheel1, size) \mapsto 27, (wheel2, size) \mapsto 27, \ldots\}$$

to the cross product obtained in (8) yields tuple $(27, 27)$ which belongs to the binary relation of $c_1 \in co(m(bike))$. In this way, we can check satisfaction of all other constraints of the instantiation in Figure 3 which are $co(m(bike)) = \{c_1, c_2, c_3\}$, $co(m(wheel1)) = co(m(wheel2)) = \{c_4, c_5\}$, and $co(m(bag1)) = co(m(bag2)) = \{c_6\}$. Due to space limitations we do not work this out in detail but by comparing Figures 2 and 3 it is easy to see that all objects in our example instantiation satisfy the constraints imposed by their underlying types. Together with the fact that the instantiation in Figure 3 is a tree with root *bike* and $m_O(bike) = Bike$ constitutes the partonomic root of the model graph, we may conclude that our example is a valid instantiation of our configuration model.

The constraint illustrated above imposed a relation on attributes with atomic values. In addition, we want to account for attributes taking aggregated values. For example, the weight of a wheel is (usually) explicitly given, while that of an entire bike must be calculated from the weights of its components. Also, we can use calculated attributes for enforcing port multiplicities, as we show below. To this end, we allow for attributes whose value is either assigned or calculated via aggregate functions, like addition or maximum. We address this via the evaluators in E along with a refinement of the valuation function v. As above, we rely on path expressions for selecting the values subject to aggregation.

Accordingly, an evaluator $e \in E$ is either $\top$, indicating that an atomic value is assigned, or a pair $((\vec{d_1}, \ldots, \vec{d_n}), f)$ where each $\vec{d_i}$ is a path expression for $1 \leq i \leq m$ and f is an aggregate function. Aggregate functions are defined on sets and yield an element from V. When the input is the empty set, this is their neutral element, e.g., 0 for the function *sum*.

Given $o \in O$, we define for $(d, e) \in at(m_O(o))$

$$v((o,d)) = \begin{cases} v \in V & \text{if } e = \top \\ f\left(\bigcup_{i=1}^{n} sel_o(\vec{d_i})\right) & \text{if } e = ((\vec{d_i})_{i=1}^{n}, f) \end{cases}$$

This function combines the assignment of attributes to atomic and calculated values.

For simplicity, we often write $d = f(\vec{d_1}, \ldots, \vec{d_n})$ whenever $(d, ((\vec{d_1}, \ldots, \vec{d_n}), f)) \in at(t)$ for some type $t \in T$. For example, in Figure 2 consider the attribute calculating the total weight of a bike indicated by *totalWeight = sum((*,weight))*. In the above notation, this corresponds to attribute $(totalWeight, e_1) \in at(Bike)$, with evaluator $e_1 = ((*, weight), sum)$. The expression *(*,weight)* is syntactic sugar for all attribute path expressions pointing to an attribute *weight*, here expanding to the sequence

$((frontWheel, weight), (rearWheel, weight),$
$(frontWheel, bag, weight), (rearWheel, bag, weight),$
$(frame, bag, weight))$.

Accordingly, the value of the calculated attribute *totalWeight* of object *bike* is 4550, the sum of two individual wheel weights 2100 and individual bag weights of 250 and 100, as shown in the instantiation in Figure 3.

The above concepts also allow us to account for port multiplicities. This can be done by using a calculated attribute constrained by all legitimate multiplicities. As an example, consider the model in Figure 2 but with the type *Wheel* extended by an attribute $\#bags$ along with the evaluator $(((bag)), count)$ and the table constraint $((\#bags), ((0), (2)))$; it expresses that a wheel must have exactly 0 or 2 bags. Instead of writing out the constraint and auxiliary attribute, we often denote this by just adding "$\{0, 2\}$" to the correponding arrow. Note that unlike above, the aggregator relies on a port path expression, yielding the bags *bag1*, *bag2* for *wheel1* and no objects for *wheel2*. Accordingly, the value of attribute *#bags* of *wheel1* (resp. *wheel2*) is 2 (resp. 0). This is among the admissible values of the constraint imposed by *Wheel*.

4. An ASP-based solution to configuration problems

For brevity, we refrain from giving an introduction to ASP. Full details on the input language of *clingo* along with various examples can be found in the *Potassco User Guide* [16].

4.1. Configuration model fact format

```
1  type((bike;wheel;frame;bag)).

3  part(bike,wheel,frontWheel).
4  multiplicity(bike,wheel,frontWheel,1).
5  part(wheel,bag,bag).
6  multiplicity(wheel,bag,bag,(0;2)).
```

Listing 1: Facts representing parts of the model graph of the bike example from Figure 2

Listings 1-4 display a snippet of the encoding representing the bike example from Figure 2. A part of the model graph is encoded in Listing 1. Types are declared via a `type/1` atom where the argument is the name of the type. Parts are declared via a `part/3` atom with source and target type and port descriptor as arguments. The corresponding multiplicites are encoded via a `multiplicity/4` atom with the same structure as the `part/3` atom plus all possible multiplicities as fourth argument.

```
1  attr(wheel,size).
2  dom(wheel,size,(22;24;27;29)).
3  attr(wheel,weight).
4  dom(wheel,weight,(1800;1900;2100;2200)).
```

Listing 2: Facts representing the *Wheel* attributes of the bike example from Figure 2

Listing 2 contains the encoding of attributes *size* and *weight* of type *Wheel*. Atomic attributes are declared via an `attr/2` atom with type and attribute descriptor as arguments together with domain `dom/3` atoms. As before, the domain atoms have the same structure as `attr/2` atoms plus the possible values as third argument.

```
1  attr(bike,totalWeight,"sum").
2  path(bike,totalWeight,
3       ((weight,(frontWheel,()));
4        (weight,(rearWheel,()));
5        (weight,(bag,(frontWheel,())));
6        (weight,(bag,(rearWheel,())));
7        (weight,(bag,(frame,())))).
```

Listing 3: Facts representing a calculated attribute of the bike example from Figure 2

Listing 3 contains the encoding of the calculated attribute *totalWeight* from type *Bike*. Calculated attributes are declared via an `attr/3` atom. The structure is the same as `attr/2` atoms plus the aggregate function as third argument (currently *sum*, *count*, *min* and *max* are supported). The corresponding path expressions which are used to gather all values are declared via `path/3` atoms. The first two arguments have to be the same as the `attr/3` atoms and the third argument is a path expression. Path expressions follow a nested tuple structure in ASP with the first element of the sequence being the innermost. Consider for example the path expression (d_1, d_2, d_3) in the formalism. We express this in ASP as `(d3,(d2,(d1,())))`. While this is not easily readable for humans, it enables one to work dynamically with tuples of any size in ASP. In the future, we plan to implement an input and output language which is human-readable and leave the nested tuple structure for internal representation.

Constraints are declared via a `constraint/1` atom where the argument is a *constraint identifier*. In Listing 4 an example of a table constraint is shown. The identifier is a tuple consisting of the type the constraint is attached to and an index. The columns of the table are declared via `column/3` atoms containing the constraint identifier, the index of the column and the path expression. The actual entries are declared via `entry/3` atoms containing the constraint identifier, a tuple with the index of the column and row, and the value of the entry.

```
1  constraint((wheel,0)).
2  column((wheel,0),0,(size,())).
3  column((wheel,0),1,(weight,())).
4  entry((wheel,0),(0,0),22).
5  entry((wheel,0),(1,0),1800).
6  entry((wheel,0),(0,1),24).
7  entry((wheel,0),(1,1),1900).
8  entry((wheel,0),(0,2),27).
9  entry((wheel,0),(1,2),2100).
10 entry((wheel,0),(0,3),29).
11 entry((wheel,0),(1,3),2200).
```

Listing 4: Facts representing a table constraint of the bike example from Figure 2

4.2. General problem encoding

The ASP encoding of our formalization can be found in Listing 5. In Lines 1-6 it is checked that the configuration model graph is indeed acyclic and rooted. Further, the type of the root object is determined which is created in Line 8 (with the correct type as second argument). In Lines 10-12 objects for each part relation are generated while making sure that the indices of the objects are assigned incrementally. Satisfaction of the multiplicites of those part relations are assured in Lines 14-15. In Lines 17-19 values are assigned to attribute values according to their domain making sure that each object has exactly one value assigned for all its attributes. All possible port and attribute selectors are created in Lines 22-24. Using selectors, the correct values for aggregates are determined and assigned. In Lines 26-27, we show the encoding for one such aggregate function *sum*. Our full implementation which can be found under `https://github.com/potassco/configuration-encoding` also contains the aggregate functions *count*, *min* and *max*. Lastly, in Lines 29-42 table constraint satisfaction is checked for each object. First, all possible tuples of the cross product are created (encoded again as nested tuples). Then the tuples are unpacked step-by-step while traversing the columns of the constraint. Only if all tuples satisfy at least one complete row, the constraint is satisfied.

Due to space limitations, we are not showing the full implementation of our formalization. As mentioned above, we are only showing the aggregate function *sum*. Additionally, we left out connection ports and comparison constraints (e.g., $==$, $\leq$, etc.) in Listing 5. In our full encoding we also distinguish between *mandatory objects* (encoded by normal rules) and *optional objects* encoded by choice rules by which we hope to achieve a better performance. In addition to that, several examples and files to visualize configuration models and instantiations using *clingraph* [17] can be found in the repository linked earlier.

```
1  partonomic_path(X,Y) :- part(X,Y,_).
2  partonomic_path(X,Z) :- partonomic_path(X,Y), partonomic_path(Y,Z).
3  :- partonomic_path(X,X).

5  root(T) :- type(T), not partonomic_path(_,T).
6  :- {root(T)} > 1.

8  object((),T) :- root(T).

10 { object((D,(O,I)),T) : I = 0..Max-1 } :-
11         object(O,S), part(S,T,D), Max = #max { N : multiplicity(S,T,D,N)}.
12 :- object((D,(O,I)),_), not object((D,(O,I-1)),_), I > 0.

14 :- part(S,T,D), object(O,S), not multiplicity(S,T,D,X),
15                     X = #count { I : object((D,(O,I)),T) }.

17 { val((O,D),V) : dom(T,D,V) } :- object(O,T), attr(T,D).
18 :- attr(T,D), object(O,T), not val((O,D),_).
19 :- val(X,V1), val(X,V2), V1 < V2.

21 attr(T,D,"atomic") :- attr(T,D).
22 selector(O,(),O) :- object(O,_).
23 selector(O,(D,P),(D,(O',I))):- selector(O,P,O'), object((D,(O',I)),_).
24 selector(O,(D,P),(O',D)) :- selector(O,P,O'), object(O',T), attr(T,D,_).

26 val((O,D),V) :- object(O,T), attr(T,D,"sum"),
27             V = #sum { V',X,P : path(T,D,P), val(X,V'), selector(O,P,X) }.

29 max_col_idx(C,N) :- constraint(C), N = #max{ Col : column(C,Col,_)}.
30 tuple((O,C),N,((),X)) :- object(O,T), max_col_idx((T,C),N),
31                          selector(O,P,X), column((T,C),N,P).
32 tuple((O,C),N,(VT,X)) :- object(O,T), tuple((O,C),N+1,VT),
33                          selector(O,P,X), column((T,C),N,P), N>=0.

35 sat_row((O,C),VT,(0,Row),VT') :-
36     object(O,T), tuple((O,C),0,VT), VT = (VT',X),
37     val(X,V), entry((T,C),(0,Row),V).
38 sat_row((O,C),VT,(Col,Row),VT'') :-
39     object(O,T), sat_row((O,C),VT,(Col-1,Row),VT'),
40     VT' = (VT'',X), val(X,V), entry((T,C),(Col,Row),V).

42 :- tuple(C,0,VT), not sat_row(C,VT,_,()).
```

Listing 5: ASP encoding for solving configuration problems.

4.3. Instantiation fact format and obtaining solutions

On the instantiation level, there are two important atoms `object/2` and `val/2`. They represent objects and the valuations of attribute variables, respectively. The `object/2` atom takes as arguments the name of the object encoded as a nested tuple and its type. The nested tuple structure is similar as for path expressions above (see Section 4.1). The names are constructed from the partonomic port descriptors and indices. Take for example the atom `object((bag,((frontWheel,((),0)),1)),bag)`. This correponds to the second bag of the first (and only) wheel with descriptor *frontWheel* of the root object (which has type *Bike*). The root is always encoded as an empty tuple `()`. Note that this way of encoding objects directly assures that the set of partonomic associations is a tree as required for valid instantiations.

The `val/2` atom takes as first argument an attribute variable encoded as a tuple. The tuple contains the object name and the attribute descriptor. The second argument of the atom is the actual value of the variable. For example, we have the atom `val(((),minStowage),30)`

which expresses that attribute *minStowage* of the root object bike has value 30.

We can run the encoding together with the file of a model M to obtain one or multiple stable models. The stable models correspond to valid instantiations as defined in Section 2. This is easy to verify, as (table) constraints are encoded as integrity constraints in ASP, thus have to be satisfied in every stable model. Further, as mentioned above our object structure directly assures that the set of partonomic associations is a tree and that the root object has the type of the partonomic root of the model graph. We can also specify user requirements U by providing, e.g. another input file `instantiation.lp`. Every instantiation obtained in form of a stable model then extends the user requirements and is therefore a solution to (M, U).

In Listing 6 we run our full encoding with the model from Figure 2. The user requirements are empty, i.e., omitted, and the solution we obtain corresponds to the one from Figure 3.

```
$ clingo encoding.lp examples/bike/model.lp
clingo version 5.6.2
Reading from encoding.lp ...
Solving...
Answer: 1
object((),bike)
object((frontWheel,((),0)),wheel)
object((rearWheel,((),0)),wheel)
object((frame,((),0)),frame)
object((bag,((frontWheel,((),0)),0)),bag)
object((bag,((frontWheel,((),0)),1)),bag)
val(((),maxWeight),5000)
val(((),minStowage),30)
val(((frontWheel,((),0)),size),27)
val(((rearWheel,((),0)),size),27)
val(((frontWheel,((),0)),weight),2100)
val(((rearWheel,((),0)),weight),2100)
val(((bag,((frontWheel,((),0)),0)),volume)
    ,20)
val(((bag,((frontWheel,((),0)),0)),weight)
    ,250)
val(((bag,((frontWheel,((),0)),1)),volume)
    ,10)
val(((bag,((frontWheel,((),0)),1)),weight)
    ,100)
val(((),stowage),30)
val(((),totalWeight),4550)
SATISFIABLE
```

Listing 6: Running the bike example from Figure 2 in *clingo*

5. Related work

Our formalism borrows concepts from various other approaches in the literature. A general ontology of configuration has been introduced in [13]. Here, a configuration problem is divided into *configuration model knowledge*, *configuration solution knowledge* and *requirements knowledge*. However, the paper argues that the latter can be expressed in terms of the other two. In the model knowledge there are *product specific classes* called types and a *configuration* of a product w.r.t. to a configuration model is defined as a set of instances of the types occurring in the model. These instances are called *individuals*. Constraints are specified inside the model and a correct configuration must satisfy these. However, the definition of constraints is left to an unspecified constraint language. A configuration also contains configuration specific relations called *properties*. Unlike our approach, [13] includes the concepts of taxonomy and inheritance.

Another formal approach to configuration in the context of constraint programming has been given by [2]. Here, a *structural configuration model* lays out the possible variations of the entity to be configured. This model contains types and attributes, as well as partonomic and connection ports. Types can be functional or technical and this restricts the possible kinds of (taxonomic) subtypes they are allowed to have. Technical types can only have concrete types as subtypes which can be seen as "complete" parts ready to be ordered from a catalog. A *configuration* can be obtained from a structural model by instantiating types. Instances inherit the attributes and ports from their type and all its supertypes. As to what regards constraints, three kinds are defined: compatibility, requirement and resource constraints.

Lastly, [14] follows a somewhat less formal, object-oriented approach at modelling configuration problems where concepts are directly defined in ASP. Again, there is a distinction between a *model* and an *instantiation*. The model contains a taxonomy of *classes* and a general association relation (with no distinction between part and connection relations). It is noteworthy, that associations in general have multiplicities in both directions. Further, attributes are limited to be over the domain of strings, integers or booleans. In an instantiation of a model, each object is defined through a global index. An "is-a" relation ties it to a class. Two objects are connected through an "associated" relation which has to correpond to an association relation from the model. *Attribute values* assign values to attributes of objects. Constraints are not specified directly but left open to general integrity constraints in ASP.

6. Discussion

We presented an approach to model and solve configuration problems and a corresponding encoding for ASP. A model and its instantiation are expressed through directed multigraphs where the former specifies the possible graphs of the latter. Similar to [2], the partonomy of the model graph has to be acyclic. We view this as

favorably for object generation.

Apart from the structural constraints imposed by the model graph, each type in the model may impose constraints on itself and its parts. For this we use table constraints as a canonical representation which specify possible combinations of attributes. In our view, table constraints are the most general form of constraints and other kinds can be expressed through them. Constraints attached to a type are checked independently for each object of that type. This guarantees that they are only applied in the correct context.

Attributes can be atomic, i.e., a value needs to be assigned, or calculated. For the latter we make use of aggregate functions. We also formalized the concepts of path expressions and selectors. The former can be composed of port or attribute descriptors and the latter return sets of attribute variables or objects. While port multiplicites are unbounded in general, we can use aggregate functions and port path expressions to restrict them.

In contrast to many other approaches, we cannot target specific objects in the instantiation with our constraints. For example, in Figure 3 it would not be possible to attach a constraint to the type *Wheel* targeting only *bag1*. This is because there is no selector starting at *wheel1* containing this bag only. Note that $sel_{wheel1}((bag))$ returns the set $\{bag1, bag2\}$ with both bags. This was done on purpose to evade symmetries. Our understanding is that if distinctions between objects are desirable, this information should be included in the model, e.g., by having separate ports as for the front and rear wheel.

In many scenarios it is desirable to configure multiple instances of the same type simultaneously. Since we require every configuration to be rooted, this might not appear possible within our approach. However, one could always add a new root type to the model, for example a *Fleet* of bikes.

A shortcoming of our formalization is that we require the attribute valuation function to be total, i.e., every instantiated attribute variable needs to have a value assigned. In user requirements, though, it might be desirable to leave certain attributes undefined.

Further, there are user requirements which cannot be expressed through an instantiation. For example, consider Figure 2 and a user who wants all bags to be of a certain color but does not care about the number of bags. This would require adding a new constraint to the model. Anyhow, we consider this to be more of a knowledge engineering problem as any such option should only be available if included in the model.

Compared to many other approaches that are tuned for practical applications, our approach is more abstract. This allows us to formally prove properties as we have demonstrated with the monotonicity of the solution space. Experience has shown that this is important when working with ASP. In the future, we intend to investigate what other properties our formalism possesses.

Partly due to this abstractness, we decided to start with a simpler approach exluding a taxonomy and thus a form of inheritance. However, we view both these concepts as vital for efficiently modelling configuration problems and we plan to extend our formalism to contain them. This would probably require the following steps:

1. Introduce *taxonomic ports* representing a "supertype" relation. Following [2], our acyclicity condition would be extended to consider these ports as well.
2. Adapt the definition of a valid instantiation. Now the root object of the model graph does not necessarily represent the object to be configured but could be specialized to a subtype. There might not even be a partonomic root anymore. In general, it should be possible to start the configuration at any node of the model graph which could be expressed through user requirements and a dedicated "root object" in the instantiation.
3. Types would inherit attributes and other types of ports from their supertypes, i.e., constraint satisfaction would have to be checked not only for the type an object is mapped to but also for all its supertypes.
4. A more difficult question is how to treat attributes and (non-taxonomic) ports which appear more than once for a set of supertypes or if this should be prohibited.

Lastly, we note that in many examples a taxonomy only appears in form of "specializations" (such as concrete types and catalogs in [2]). This feature can be represented in our formalism by adding table constraints (see for example constraint $c4$ in Figure 2 describing the possible wheels).

Further, we plan to further study connection ports by working out examples. They are part of the literature [2] and we view them as an important complement to partonomic ports since they convey additional information and allow to form cycles in the model graph. Consider the configuration of a computer network. Here, a configuration might have identical parts like switches which can be connected in numerous ways. We also intend to investigate symmetry conditions for connection ports which currently do not seem to be expressible within our formalization of constraints.

On the implementation level, our full encoding currently supports table and comparison constraints. In the literature other kinds such as requirement and incompatibility constraints often occur. We plan to study how these can be represented in our formalism and to add them to our implementation.

Acknowledgments

This work was partially funded by ZIM (Zentrales Innovationsprogramm Mittelstand) of the German Federal Ministry for Economic Affairs and Climate Action (grant number: KK5291302GR1). We thank Joachim Baumeister, Richard Comploi-Taube, Andreas Falkner, Konstantin Herud, Jochen Reutelshöfer and Gottfried Schenner for their valuable feedback.

References

[1] R. Cunis, A. Günter, H. Strecker, Das PLAKON-Buch, volume 266 of *Informatik Fachberichte*, Springer-Verlag, 1991. doi:10.1007/978-3-662-06485-6.

[2] U. Junker, Configuration, in: F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier Science, 2006, pp. 837–873. doi:10.1016/s1574-6526(06)80028-3.

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

[4] G. Brewka, T. Schaub, Zur Verwendung nichtmonotoner Inferenztechniken bei der Konfiguration, in: F. di Primio (Ed.), Methoden der Künstlichen Intelligenz für Grafikanwendungen, Addison-Wesley, 1995, pp. 45–60. In German.

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

[6] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: From theory to practice, Artificial Intelligence 187-188 (2012) 52–89. doi:10.1016/j.artint.2012.04.001.

[7] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012. doi:10.1007/978-3-031-01561-8.

[8] M. Gebser, R. Kaminski, T. Schaub, aspcud: A Linux package configuration tool based on answer set programming, in: C. Drescher, I. Lynce, R. Treinen (Eds.), Proceedings of the Second International Workshop on Logics for Component Configuration (LoCoCo'11), volume 65 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 2011, pp. 12–25. doi:10.4204/eptcs.65.2.

[9] A. Felfernig, A. Falkner, M. Atas, S. Erdeniz, C. Uran, P. Azzoni, ASP-based knowledge representations for IoT configuration scenarios, in: L. Zhang, A. Haag (Eds.), Proceedings of the Nineteenth International Configuration Workshop (CONF'17), 2017, pp. 62–67.

[10] E. Gençay, P. Schüller, E. Erdem, Applications of non-monotonic reasoning to automotive product configuration using answer set programming, Journal of Intelligent Manufacturing 30 (2019) 1407–1422. doi:10.1007/s10845-017-1333-3.

[11] J. Tiihonen, M. Heiskala, A. Anderson, T. Soininen, WeCoTin – A practical logic-based sales configurator, AI Communications 26 (2013) 99–131. doi:10.3233/aic-2012-0547.

[12] J. Tiihonen, A. Anderson, VariSales, in: [3], 2014, pp. 309–318. doi:10.1016/B978-0-12-415817-7.00026-8.

[13] T. Soininen, J. Tiihonen, T. Männistö, R. Sulonen, Towards a general ontology of configuration, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12 (1998) 357–372. doi:10.1017/s0890060498124083.

[14] A. Falkner, A. Ryabokon, G. Schenner, K. Shchekotykhin, OOASP: connecting object-oriented and logic programming, in: F. Calimeri, G. Ianni, M. Truszczyński (Eds.), Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15), volume 9345 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 2015, pp. 332–345. doi:10.1007/978-3-319-23264-5_28.

[15] L. Hotz, A. Felfernig, M. Stumptner, A. Ryabokon, C. Bagley, K. Wolter, Configuration knowledge representation and reasoning, in: [3], 2014, pp. 41–72. doi:10.1016/b978-0-12-415817-7.00006-2.

[16] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, Potassco User Guide, 2 ed., University of Potsdam, 2015. URL: http://potassco.org.

[17] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, clingraph: ASP-based visualization, in: G. Gottlob, D. Inclezan, M. Maratea (Eds.), Proceedings of the Sixteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22), volume 13416 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 2022, pp. 401–414. doi:10.1007/978-3-031-15707-3_31.