Solving configuration problems with unbounded cardinalities in COOM: Preliminary report

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

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

Abstract

Industrial product configuration often involves complex partonomic structures with intricate, interdependent constraints such that the required number of components is unknown in advance. This paper addresses the challenge of determining minimal bounds for unbounded cardinalities within the CoomSuite workbench. While previous methods relied on a "naive" single-shot approach, requiring grounding and solving from scratch for every iteration, we introduce an extended workflow that leverages *clingo*'s multi-shot capabilities. We evaluate our approach across three benchmark domains demonstrating the efficiency of our incremental method compared to traditional single-shot strategies in scenarios with unbounded cardinalities.

1. Introduction

Industrial product configuration [1, 2] problems typically have large partonomic structures with complex interdependencies. While some cardinalities are fixed, e.g., a bike has 2 wheels, others are dynamic and depend on constraints and requirements provided by the user. This impedes determining the exact (or even approximate) number of objects needed in advance [3]. As an example, consider a customer who wants to order a new bottling plant with a required minimal capacity of bottles per minute. Configuring this bottling plant consists of generating a number of instances of bottling robots together with parallel instances of conveyor belts matching the robots. Finding a suitable minimal number of robots and conveyor belts or even an estimation of lower and upper bounds is not trivial as the capacity of each robot depends on various types of factors and constraints. Additionally, constraints between robot instances might further restrict their capacity.

We started addressing this challenge in [4] in the context of the CoomSuite [5], a workbench for experimenting with industrial-scale product configuration problems formulated in the industrial configuration language Coom [6].[1] The CoomSuite contains various benchmark sets, Coom grammar definitions, and further resources related to Answer Set Programming (ASP; [7]). While our basic workflow from [4] only allowed for solving problems with bounded cardinalities, we introduced a simple method for unbounded ones with two possible search strategies to determine minimal bounds. However, this approach does not make use of *clingo*'s multi-shot capabilities [8], and thus required grounding and solving from scratch at each iteration. We address this limitation here by

1. introducing an extended workflow for the CoomSuite with a new, general ASP multi-shot encoding, and
2. evaluating this workflow on three benchmark domains (*CargoBike*, *TravelBikeFleet*, *Restaurant*).

Note that the *TravelBikeFleet* and *Restaurant* domains were already evaluated in [4] in a non-incremental, single-shot setting and have been adapted to contain unbounded cardinalities. This allows us to compare

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026) co-located with the Federated Logic Conference (FLoC 2026).

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

[1] Coom is developed by denkbares and used in numerous industrial applications.

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

CEUR-WS.org/Vol-4311/ASPOCP_regular_06.pdf

our incremental approaches with the non-incremental one and shows that even when upper bounds exist, it can be beneficial to solve a problem incrementally.

The remainder of our paper is structured as follows: We start by providing the required background in Section 2 and continue by introducing our multi-shot encoding in Section 3. Finally, we show and discuss the results of our empirical evaluation in Section 4.

2. Background

In this section, we review all the necessary parts for introducing our multi-shot encoding in the next section. We start by giving a brief overview of *clingo*'s multi-shot solving capabilities [8, 9] and then introduce the CoomSuite [4].

2.1. Multi-shot solving in *clingo*

Multi-shot solving allows for solving continuously changing logic programs in an operative way through looping on grounding and solving while reacting, for instance, to any outside changes or previous solving results. This does not only avoid re-grounding previously added parts but also benefits solving through constraints learned over time.

The basic building blocks needed for this are the possibility to split a (non-ground) logic program into subprograms (by means of the `#program` directive) and so-called *externals* (by means of the `#external` directive), which allow us to activate and/or deactivate ground rules on demand.

For example, the program in Listing 1 introduces a new program part called `part` with a single parameter `b`. All rules, external statements, and show statements up to the next `#program` directive or the end of the file belong to the program part.[2] The *clingo* API allows us to specify a specific value for the parameter `b` and to ground the part separately from other rules of the program. Next, Line 2 declares an external atom `active(b)` which uses the local parameter `b`. During grounding, this parameter is replaced with the specific value provided for `b`. The truth value of external atoms can be set from "outside of the program" using the Python API.[3] This allows us to control whether the rule in Line 3 is active or not by setting the truth value of the external atom to true or false, respectively.

```
1 #program part(b).
2 #external active(b).
3 fact(b) :- active(b).
```

Listing 1: Example of program and external directives

2.2. The CoomSuite

2.2.1. The Coom language

Our language of choice for representing product configuration models is Coom. In [4], we distinguished between three Coom language fragments of increasing complexity plus corresponding extensions for two of them. We focus here on the language fragment Coom[x*] represented by the *CargoBike* example in Listing 2. This fragment is the unbounded extension of Coom[x], which allows for modelling various features like partonomies and a broad range of (mostly linear) numerical calculations and constraints.

The *CargoBike* has two features: The first one, `requestedVolume`, is numeric with range `0-200` and serves as input to the configuration problem (Line 2). The second one represents an unknown number of needed `bags` (Line 3) expressed by the cardinality `0..*` where `*` represents an unknown upper bound. The type `Bag` (Line 5-7) is a *structure* with a single feature `Size`. Structures in Coom represent parts, and thus allow for building complex partonomies. The type `Size` is an *enumeration*, which represent discrete attributes, and provides a number of different options (`Small`, `Medium`, `Large`), each with an

[2] Statements not preceded by a `#program` directive implicitly belong to the `base` program part.
[3] For more information, see the Potassco User Guide [10].

```
1  product {
2              num  0-200   requestedVolume
3      0..*    Bag          bags
4  }
5  structure Bag {
6              Size         size
7  }
8  enumeration Size {
9      attribute num weight
10     attribute num volume

12     Small   = ( 10 12 )
13     Medium  = ( 15 16 )
14     Large   = ( 25 20 )
15 }
16 behavior {
17     require sum(bags.size.volume) >= requestedVolume
18 }
```

Listing 2: Representation of the *CargoBike* example in Coom

associated value for the `weight` and `volume` (Lines 8-15). For instance, a `Bag` with `size` `Small` has a `weight` of 10 and a `volume` of 12. Finally, the constraint in Line 17 ensures that the total `volume` of all `bags` is greater or equal to the `requestedVolume`.

2.2.2. Solving unbounded Coom product models incrementally

The workflow for solving Coom product models with unbounded cardinalities is depicted in Figure 1.[4]

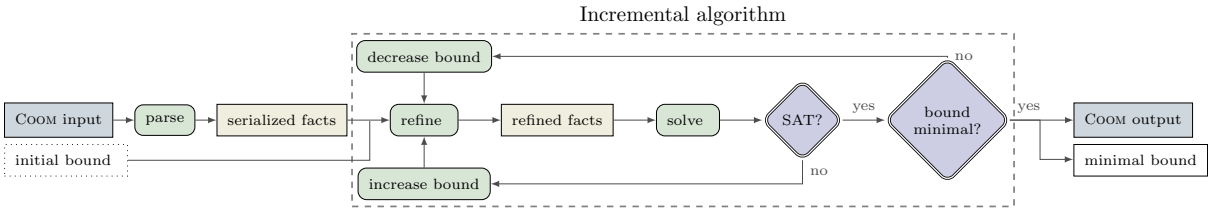


Figure 1: Workflow for solving an unbounded Coom model incrementally

The input to the problem is a Coom product model and possibly an initial upper bound (0 by default). The output, if it exists, consists of a Coom configuration, i.e., a solution to the underlying configuration problem, plus the corresponding minimal upper bound. Since Coom product models can be unsatisfiable, the loop in Figure 1 might not terminate.

Regarding the workflow, first, the Coom product model is parsed and converted into a "serialized" ASP fact format that closely resembles the Coom language. The unspecified upper bound `*` (cf. Line 3 in Listing 2) is translated into *clingo*'s `#sup`[5] expression and used to automatically detect if an input model needs to be solved incrementally. Then, the serialized facts are processed using the current bound into a "refined" ASP fact format that is independent of Coom and captures the underlying configuration problem. Note that the bound does not necessarily equal the upper bound of unbounded cardinalities. Rather it is used to determine the range for any unbounded cardinality, e.g., for the bound n an unbounded cardinality `k..*` is replaced by `k..k+n`. This initiates a loop where, first, the upper bound is increased until the problem becomes satisfiable and, subsequently, decreased (and possibly increased again) until it converges to the minimal upper bound. For the incremental "single-shot"

[4]For more information on the general workflow, see [4].
[5]The `#sup` expression is a *clingo* built-in representing the greatest element among all variable-free terms, cf. [10].

implementation this requires that the entire problem has to be refined and ground from scratch for every new bound. Lastly, the answer sets representing valid configurations are transformed back into Coom solutions. Currently, the CoomSuite offers two search methods to determine the minimal upper bound: A linear and an exponential one [11], both of which are parametrizable by the user.

Given an *initial bound* β_i, the linear search method increases this bound by a fixed *step size* s. The exponential search method, in contrast, has a *base* parameter b from which a sequence $\beta_0, \beta_1, \beta_2, \ldots$ is obtained with $\beta_k = \lceil b^k \rceil$. Note that the b^k are rounded up because b is not restricted to be an integer value. The search starts with the initial bound β_i and continues with β_{k_0}, where k_0 is the smallest value such that $\beta_{k_0} > \beta_i$. Subsequently, we continue with the β_k sequence, skipping any repeated values.

The defaults values for the step size s and the base parameter b are 1 and 2, respectively.

When an upper bound β_u has been found for which the problem is satisfiable, it is not necessarily minimal. Therefore, we recursively converge β_u to a minimal bound β_m using binary search within the interval $(\beta_l, \beta_u]$, where β_l is the previous bound (for which the problem is unsatisfiable). We compute the (integer) midpoint $\beta = \lfloor (\beta_u + \beta_l)/2 \rfloor$ and solve at this bound. If the problem is satisfiable with bound β, then β becomes the new upper bound, otherwise β becomes the new lower bound. The search terminates once β_l and β_u are consecutive integers. Note that this strategy is the same for both search methods independent of the chosen parameters.

2.2.3. Solving Coom models with ASP

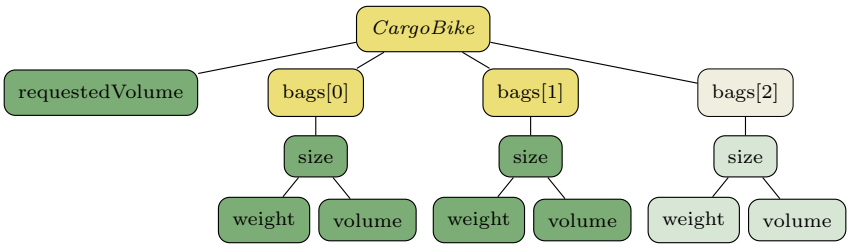


Figure 2: The *CargoBike* example converted into a configuration tree

Fact format For the sake of brevity and as done in [4], we omit the "serialized" ASP fact format as it does not provide any further insights and only give a brief overview of the intuition behind our "refined" ASP representation, as well as an introduction to the predicates needed for Section 3.

The essential concepts of our "refined" fact format are a partonomy with bounded cardinalities, attributes ranging over discrete and integer domains, and Boolean as well as table constraints, where the former allow for arithmetic expressions and aggregate functions. This representation does not allow for unbounded cardinalities, i.e., all possible objects need to be stated explicitly, and thus an upper bound needs to be specified beforehand (compare with Figure 1).

We represent a configuration as a tree such that each node represents a variable, and its root reflects the object to be configured.[6] Variables can be either parts or attributes. Solutions are represented as subtrees of this tree where values are assigned to any attribute variable included in the subtree. Thus, not all variables are necessarily included in the solution and an excluded variable renders all variables in its subtree excluded. Cardinalities are represented as constraints over sets of nodes in the tree.

The configuration tree for the *CargoBike* is shown in Figure 2. This can be seen as an explicit representation of the Coom product model of Listing 2 where we assumed an upper bound of 3 for the `bags`. Nodes belonging to parts are highlighted in yellow and those belonging to attributes in green. Consider an example solution where the third `bag` is undefined, thus not included in the solution. We represent this in Figure 2 by highlighting the corresponding variables in a lighter color. Note that undefined variables automatically render their subnodes undefined as well.

[6]For a formal approach of this representation we refer the reader to [12].

The structure of the configuration tree serves as the basis for our fact format, where all possible variables within the model are generated. For each variable, a unique (human-readable) identifier is assigned by leveraging the tree-like structure of the configuration. For example, the `volume` feature of the second instance of the `bags` feature in Listing 2 results in the variable `"bags[1].size[0].volume[0]"`. This is captured in ASP by three facts representing the type, index and parent of the variable as shown in Listing 3 for the example. As mentioned above, this requires that an upper bound for every variable has to be provided.

```
1 type("bags[1].size[0].volume[0]","Size.volume").
2 index("bags[1].size[0].volume[0]",0).
3 parent("bags[1].size[0].volume[0]","bags[1].size[0]").
```

Listing 3: A variable of the instantiated *CargoBike*

Boolean constraints, which include arithmetic formulas and aggregates, are encoded recursively in binary and unary formulas. Listing 4 shows the `require` constraint from Listing 2. First, the constraint is instantiated using a `constraint/2` predicate, where the first argument is the constraint identifier consisting of an index and the entire formula as a string, and the second argument denoting the type of the constraint. Note that this could be any unique string but we opt for a human-readable format.

The formula of the constraint, which has operator $\geq$, is separated into its two sub-formulas by using a `binary/4` predicate. Next, the `sum` aggregate is encoded using a `function/3` predicate, where the first argument is the identifier of the entire aggregate, the second argument denotes its type, and the third argument denotes the set on which the aggregate is being applied. Lastly, the members of the aggregate set are encoded using `set/2` predicates. Following our example from Figure 2, we assume that an upper bound of 3 has been provided.

```
1 constraint((0,"sum(bags.size.volume)>=requestedVolume[0]"),"boolean").
2 binary("sum(bags.size.volume)>=requestedVolume[0]",
3        "sum(bags.size.volume)",">=","requestedVolume[0]").
4 function("sum(bags.size.volume)","sum","bags.size.volume").
5 set("bags.size.volume","bags[0].size[0].volume[0]").
6 set("bags.size.volume","bags[1].size[0].volume[0]").
7 set("bags.size.volume","bags[2].size[0].volume[0]").
```

Listing 4: The constraint of the instantiated *CargoBike*

***clingo* encoding** Solutions are captured in terms of `include/1` predicates, stating that a variable of the configuration tree is included in the solution, and `value/2` predicates, assigning values to attribute variables. Both are generated via the choice rules in Listing 5, where the fact `include("root")` and the integrity constraint in Line 2 ensure that each solution is a subtree of the configuration tree starting at its root.

```
1 { include(X) : type(X,_) }. include("root").
2 :- include(X), parent(X,P), not include(P).
3 { value(X,V) : domain(T,V) } = 1 :- include(X), type(X,T), attribute(T).
```

Listing 5: Definedness of variables and generation of attribute values

Listing 6 shows the rules necessary to determine whether the constraint in Listing 4 is satisfied. For the `sum` calculation, we use *clingo*'s built-in `#sum` aggregates. Then, the integrity constraint in Line 3 enforces that a constraint cannot be violated. This happens if it is defined and its formula is not satisfied. For the sake of brevity, we omit the rules regarding definedness but, intuitively, a formula is defined if all variables in it have a value assigned.

```
1 value(F,V) :- function(F,"sum",P), V = #sum{ V',X : set(P,X), value(X,V')}.
2 satisfied(F) :- binary(F,XL,">=",XR), VL >= VR, value(XL,VL), value(XR,VR).
3 :- constraint((ID,F),"boolean"), defined(F), not satisfied(F).
```

Listing 6: Subset of constraint checking rules

***flingo* encoding** An alternative encoding is provided using hybrid solver *flingo* [13], which is a prototype system for solving conditional linear constraints with integer variables in ASP. *flingo* uses CASP solver *clingcon* [14] as a backend, and therefore, is able to deal with large integer ranges and numerical calculations. The main difference, however, is that variables in *flingo* can stay undefined, while in *clingcon* all integer variables need to have a value assigned. As Coom allows for optional attributes, this is a crucial feature.

The encoding for *flingo* is identical in many parts to the *clingo* encoding, except for rules which treat integer variables and numerical constraints. For the sake of brevity we only show a few illustrative excerpts.

Listing 7 shows the rule for generation of integer attribute values. The theory atom `&in{L..U} =: X` in the head assigns a value from the interval `L..U` to variable `X`. The operator `=:` specifies that this is an *assignment*, meaning that `X` only gets a value assigned if the lower and upper bound of the interval are defined. Note that for integer variables we do not need to use the `value/2` predicate anymore since *flingo* assigns their values directly to the corresponding terms; however, *flingo* uses this same predicate for its output, thus is consistent with our fact format.

```
1 &in{L..U} =: X :- include(X), type(X,T), integer(T), range(T,L,U).
```

Listing 7: Generation of integer attribute values

Regarding constraints, Listing 8 shows the adapted rules from Lines 1 and 2 in Listing 6. The main difference in Line 1 is that we make use of *flingo*'s `&sum` atom instead of the `#sum` aggregate, and that we can move this atom to the head as we do not need the `value/2` predicate for integer variables. Note also that in Line 2 we need the `&sum` to access the values of the integer variables. This is because terms which correspond to integer variables are treated differently internally.

```
1 &sum{ X : set(P,X) } = F :- function(F,"sum",P).
2 satisfied(F) :- binary(F,XL,">=",XR), &sum{XL} >= XR.
```

Listing 8: Subset of constraint checking rules in *flingo*

Coom solution Consider now a scenario, where the user runs the CoomSuite with the user input `requestedVolume[0]=35`, and the linear search algorithm with a step size of 3. The corresponding command-line output, shortened for the sake of readability, is shown in Listing 9. Clearly, the program is unsatisfiable for the initial bound 0. It is satisfiable for the next bound 3 but this bound is not yet (proven to be) optimal. Subsequently, the converge algorithm tries out the intermediate bounds 1 and 2 following the procedure described above, and determines that the latter is a minimal, satisfiable bound. Then, a solution for the minimal bound, converted back into Coom, is printed.

```
Solving with max_bound = 0: UNSATISFIABLE
Solving with max_bound = 3: SATISFIABLE     Optimal bound not yet found
Solving with max_bound = 1: UNSATISFIABLE   Optimal bound not yet found
Solving with max_bound = 2: SATISFIABLE     Optimal bound found

    requestedVolume[0]=35
    bags[0]
        bags[0].size[0]=Large
        bags[0].size[0].weight[0]=25
        bags[0].size[0].volume[0]=20
    bags[1]
        bags[1].size[0]=Medium
        bags[1].size[0].weight[0]=15
        bags[1].size[0].volume[0]=16

    The minimal upper bound is 2.
```

Listing 9: Command-line output of the incremental CoomSuite workflow

3. A multi-shot approach to solving Coom[x*]

As mentioned in the previous section, solving unbounded cardinalities using the "single-shot" approach involves preprocessing as well as grounding and solving the entire problem from scratch for every new bound. We now discuss an approach making use of *clingo*'s multi-shot capabilities. We start by introducing the CoomSuite multi-shot workflow and then present some illustrative program parts of the ASP encoding.

3.1. Workflow

Algorithm 1 shows a procedure, impliemented via *clingo*'s Python API, that for every increase of the maximum upper bound only adds the newly created objects and their corresponding atoms. This procedure is very similar to the workflow of Figure 1; however different from before, we construct the ground program step-by-step for every new bound.

Algorithm 1: Coom multi-shot Solving

```
Input: serialized_facts
Input: initial_bound
Output: minimal_bound
SAT ← False;
ground_program ← { };
bounds ← (initial_bound, initial_bound);
while not SAT do
    ground_program ← ground_program ∪ update_program(serialized_facts, bounds);
    SAT ← solve(ground_program);
    if SAT then
        minimal_bound ← converge_bound(ground_program, bounds);
    else
        bounds ← (bounds[1], next_bound());
```

The main differences with the workflow in Figure 1 are contained in function *update_program*, which is shown in Algorithm 2. Its input are the serialized ASP facts as well as a tuple containing the previous and current upper bound. Its output is a ground program that is added to the current ground program in the main loop. Even though our supported algorithms may increase the bound in larger steps, the addition of program parts and grounding is done in increments of 1. However, solving is only done for the bounds provided by the search algorithm (cf. Algorithm 1).

For each bound increment, we first process the serialized facts. The resulting refined facts are then filtered to remove any facts already obtained in previous steps. Next, the program parts corresponding to each of the newly refined facts are collected and grounded. The values of externals are updated to activate/deactivate certain rules. We discuss the specific function of each of the externals, `active` and `max_bound`, below when introducing the multi-shot encoding.

The convergence function `converge_bound` (cf. Algorithm 3 in Appendix A) implements the previously discussed binary search between a lower (unsatisfiable) and upper (satisfiable) bound. During convergence we do not have to add any new parts to the program but merely adjust which parts of the program are active by changing the truth values of externals.

3.2. Encoding

As not all facts and rules need to be treated in an incremental way, we split our ASP encoding and the corresponding fact format into two parts. First, a "base" part containing all facts and rules which

Algorithm 2: Update program

Input: serialized_facts
Input: bounds
Output: ground_program

```
1  ground_program ← {};
2  foreach bounds[0] < b ≤ bounds[1] do
3      refined_facts ← refine(serialized_facts);
4      new_facts ← filter(refined_facts);
5      program_parts ← {};
6      foreach fact ∈ new_facts do
7          program_parts ← program_parts ∪ get_program_part(fact, b);
8      ground_program ← ground_program ∪ ground(program_parts);
9      Set external active(b) to True;
10 Set external max_bound(bounds[0]) to False;
11 Set external max_bound(bounds[1]) to True;
```

correspond to variables with fixed, thus bounded, cardinalities,[7] and second, a multi-shot part containing facts and rules corresponding to variables with unbounded cardinalities.

This second part consists again of various program parts, one for each type of fact that can be added by an increased bound. These program parts were obtained by partitioning the rules of the single-shot encoding and adding some small modifications such as including externals. For reasons of space, we only demonstrate a few of these program parts together with their necessary modifications by the following example scenario.

Consider that our previous upper bound was 3 (cf. Figure 2) and that our new upper bound is 4. The new complete configuration tree is depicted in Figure 3.

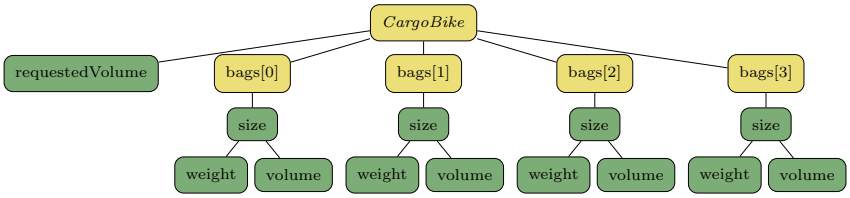


Figure 3: The *CargoBike* example converted into a configuration tree with maximum upper bound 4

3.2.1. Variables

While, in general, a bound increase can add an arbitrarily complex subtree with many objects, here, the new variables are `"bags[3]"` and its corresponding attributes, e.g., `"bags[3].size[0]"` and so forth. These variables are added to the program using the program part shown in Listing 10 (cf. Listing 5 for the single-shot case). Note that we present simplified versions of the encodings here to illustrate the concepts.

Line 1 defines the program part and its parameters: the `name` and `type` of the variable as well as the current `bound`. Line 2 defines an external to deactivate variables introduced at some specific bound. As noted above, this is required for decreasing the bound while converging to the minimal bound.

Next, Line 3 defines a new fact recording the `bound` at which a variable was introduced. This auxiliary predicate is used for the evaluation of constraints later on. The remaining three rules correspond to the rules in Listing 5. A fact `type/2` represents the variable and its type and the two choice rules control whether the variable is included in the configuration and which value it receives (if it is an attribute).

[7] This part corresponds to the fact format and encoding from [4] which we partially discussed in Section 2.2.

Note the inclusion of the external `active(bound)` in the body of the choice rule in Line 5 to prevent the inclusion of a variable when its corresponding bound is not active.

```
1 #program new_variable(name,type,bound).
2 #external active(bound).
3 bound(name,bound).
4 type(name,type).
5 { include(name) } :- active(bound).
6 { value(name,V) : domain(type,V) } = 1 :- include(name), type(name,type), attribute(type).
```

Listing 10: Simplified program part for adding new variables.

3.2.2. Constraints

We differentiate between *simple* and *incremental* constraints. While the truth values of the former are not affected by additions of variables for higher bounds, the evaluation of the latter needs to be updated with each new bound.

As an example of a simple constraint, consider the extension of the *CargoBike* from Listing 2 by adding a color feature to the bags as shown in Listing 11.

```
1  ...
2  structure Bag {
3              Size         size
4              Color        color
5  }
6  enumeration Color { Green Blue Red }
7  ...
8  behavior Bag {
9      require color = Blue
10 }
```

Listing 11: Extension of the *CargoBike* example with colors for bags.

The constraint in Line 9 has to be added for each new instance of the type `Bag`; however, it is not affected by additional instances of `Bag` later on. Therefore, the addition of such a constraint is handled similarly to the addition of new variables discussed above. Importantly, the external `active(bound)` is used to enforce the constraint only while its corresponding bound is active.

On the other hand, consider the constraint in Line 17 of Listing 2. This constraint involves the aggregate `sum(bags.size.volume)`, thus reasons over the set of all `bags`. While it is only added once to the problem, its evaluation needs to be updated with each new instance of the type `Bag`.

We define incremental constraints or, more generally, incremental expressions recursively. A *set* is incremental if its associated cardinality is unbounded. A (set-based) *function* is incremental if its set argument is incremental. A *unary* or *binary* expression is incremental if it has a sub-expression which is incremental and a *Boolean constraint* is incremental if its corresponding expression is incremental.[8]

Returning to our example scenario, after increasing the bound to 4, we would only obtain the new fact `set("bags.size.volume","bags[3].size[0].volume[0]")` for our constraint. All other facts from Listing 4 had already been obtained for bound 3.

To record the value (or satisfaction) of an expression in our single-shot encoding, we used the predicates `value/2` (or `satisfied/1`) (cf. Listing 6). However, for incremental expressions the truth values of these predicates now additionally depend on the bounds for which they were evaluated as for each bound the underlying set of elements can change. As *clingo* does not allow the redefinition of existing atoms [8], we instead introduce new `value/3` and `satisfied/2` predicates which have an additional bound parameter to record the value (or satisfaction) of an expression at a specific bound. Listing 12 shows some of the program parts for defining these new predicates (cf. Listing 6).

For incremental functions we define the value at a new bound using the value at the old bound. For example, the value of the `sum` function gets defind at bound `n+1` by taking its value at bound `n`

[8] Coom also supports tables constraints whose treatment we omit but which are fully supported by the multi-shot encoding.

and adding the values of all objects introduced at bound n+1 that belong to the underlying set. In our example scenario only a single element is added to the set, however, in general an arbitrary number of objects may be added by a new bound.

For program part update_incremental_function, this addition is expressed by the rule in Lines 7 and 8. Here, we make use of the auxiliary bound/2 predicate recording at which bound an object was introduced (see Listing 10). For "initializing" a new incremental function, we have a separate program part, new_incremental_function, which does not refer to any previous values of the function.

Any incremental expression then refers to the value of its incremental sub-expression(s) by means of the value/3 (or satisfied/2) predicate and in turn its own value is recorded using value/3 (or satisfied/2). Furthermore, for binary expressions we need different program parts depending on whether only the left, only the right, or both sub-expressions are incremental. For instance, the expression "sum(bags.size.volume)>=requestedVolume[0]" has an incremental expression on the left side only. The corresponding rule checking its satisfaction is shown in Lines 11-13 and belongs to program part incremental_binary_l. Here, the value of the left sub-expression is fetched using a value/3 predicate, while the right sub-expression is taken from a value/2 predicate.

```
1  #program new_incremental_function(f,agg,p,bound).
2  function(f,agg,p).
3  value(f,V,bound)  :- function(f,agg,p), agg = "sum",
4                       V = #sum{ V',X : set(p,X), value(X,V') }.

6  #program update_incremental_function(f,agg,p,bound).
7  value(f,V+V',N+1) :- function(f,agg,p), agg = "sum", bound = N+1, value(f,V,N),
8                       V' = #sum{ Val,X : set(p,X), bound(X,bound), value(X,Val) }.

10 #program incremental_binary_l(f,xl,op,xr,bound).
11 binary(f,xl,op,xr).
12 satisfied(f,bound) :- binary(f,xl,op,xr), value(xl,VL,bound), value(xr,VR),
13                       VL >= VR, op = ">=".
```

Listing 12: Subsets of the program parts for incremental expressions

Finally, we consider the rules for incremental Boolean constraints shown in Listing 13 (cf. Listing 6). Line 1 defines the program part with parameters for the constraint identifier cons as well as the current bound. Line 2 defines the external max_bound. This external is similar to active defined in Listing 10, however, active is true for every bound that is currently active (i.e. if the current bound is n, then active holds for $n, n-1, \ldots, 0$), whereas max_bound is only true for the current maximum bound and false for all others. This is because incremental constraints only need to hold for the current bound and not for earlier bounds. Line 4 forbids the violation of a constraint as long as its associated bound is the maximum bound.

```
1  #program incremental_constraint(cons,bound).
2  #external max_bound(bound).
3  constraint(cons).
4  :- constraint(cons), cons = (C,F), max_bound(bound),
5     defined(F,bound), not satisfied(F,bound).
```

Listing 13: Program part for incremental constraints.

4. Benchmarks

We evaluate the multi-shot encoding on three different benchmark domains[9] (cf. Table 1) that are representative of typical industrial problems. That is, in a nutshell, that some abstract structure is given over which attributes and constraints create non-trivial dependencies. Furthermore, the solution and exact number of actual instances required depends on some (user-given) requirement.

[9] https://github.com/potassco/coom-benchmarks

Table 1
Incremental benchmark sets of the CoomSuite

Benchmark set	Language	Input	Scalable factor	# Instances
CargoBike	Coom[x*]	*minVolume*	#Bags	25
TravelBikeFleet	Coom[x] & Coom[x*]	*minVolume*, *maxPrice*	#Bikes	20
Restaurant	Coom[x] & Coom[x*]	*totalSeats*	#Tables	15

While the *CargoBike* benchmark domain corresponds to an extended version of the example from Listing 2, the other two benchmark domains, *TravelBikeFleet* and *Restaurant*, were already part of the benchmark sets evaluated in [4].[10] The configuration tasks of the *TravelBikeFleet* and *CargoBike* are similar but differ in a few details. First, the *TravelBikeFleet* configures a (possibly unbounded) number of bikes that each allows a limited number of bags, while the *CargoBike* configures only a single bike with an unbounded number of bags. Both problems take as input a requested minimal volume, however, for the *TravelBikeFleet* the user additionally can specify a maximum total price that must not be exceeded. Thus, the *TravelBikeFleet* instances do not only contain more arithmetic formulas but also can be unsatisfiable if a too low maximum price has been set. We adapted the parameters in the instance generation such that we allow for around 10% of unsatisfiable instances. Lastly, for the *Restaurant* domain, the aim is to configure tables with differing number of chairs such that we obtain an exact total number of requested seats.

However, since in [4], only bounded versions in Coom[x] existed, we created incremental versions in Coom[x*] by adapting the respective cardinalities to unbounded ones. Note that the change to unbounded cardinalities adds an implicit minimization on the number of bikes (which could also have an effect on the solving time). Note that we do not use the other two benchmarks from [4], *Core* and *CityBikeFleet*, as the first one only contains cardinalites of the form `1..1` and the second one already has the number of required bikes given as input to the problem. Representative Coom models of all three benchmark sets from Table 1 are shown in Appendix B.

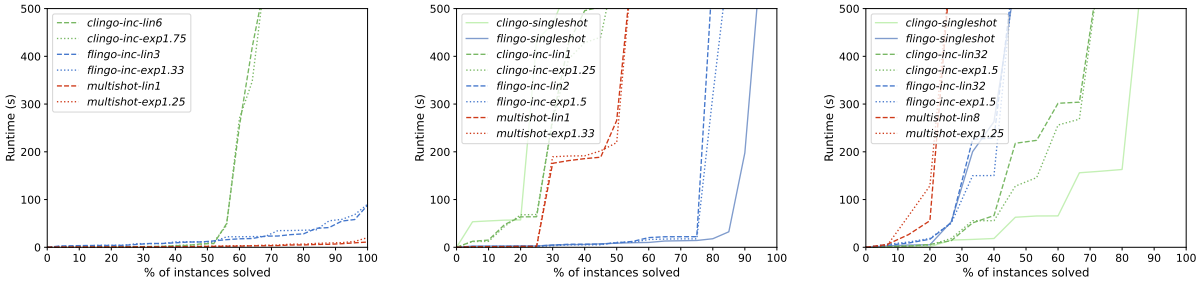
We ran all benchmark instances on a compute cluster with Intel Xeon E5-2650v4@2.9GHz CPUs with 64GB of memory running Debian Linux 10.[11] We used a timeout of 10 minutes and limited the memory to 20GB per instance. Furthermore, we used version v1.1.2 of the CoomSuite [5].

The bounded instances (Coom[x] versions of *TravelBikeFleet* and *Restaurant*) were run using the *clingo* and *flingo* single-shot encodings from [4]. All unbounded instances were run with both the naive "single-shot" incremental approach (cf. Section 2) and the multi-shot encoding presented in Section 3. For the rest of the paper we refer to the single-shot approaches as *single-shot clingo*, resp. *flingo*, and to the naive "single-shot" incremental approaches as simply *incremental clingo*, resp. *flingo*. The multi-shot encoding only works with *clingo* and is referred to as *multishot*. For all unbounded instances, we also evaluated the linear and exponential search algorithms with different parameters. More precisely, for the linear algorithm we used various integer step sizes in the range of 1 to 32, and for the exponential algorithm exponential bases in the range of 1.25, to 2.75. For the latter, as described in Section 2.2, the bound values are rounded up to obtain integer values and repeated bounds are skipped. We do not consider larger exponential bases as they performed worse in preliminary experiments.

The results per domain are presented in Figure 4. For the sake of readability, we only show the best performing parameter for each combination of incremental approach (*clingo*, *flingo*, multishot) and search algorithm (linear, exponential). This selection was performed based on the minimum average runtime of all instances per parameter.

The results show that the performance of the different encodings strongly depends on the problem domain. For the *CargoBike*, the multi-shot encoding outperforms both incremental *clingo* and *flingo*, solving all instances within a few seconds. Meanwhile, incremental *flingo* has comparable but slightly slower runtimes and incremental *clingo* is only able to solve about two thirds of the instances within the time limit. For the *TravelBikeFleet*, the multi-shot encoding is clearly slower than single-shot and

[10] Note that some small adjustments were made to the instances for the sake of having a more balanced benchmark set.
[11] https://www.cs.uni-potsdam.de/bs/research/labs.html#hardware



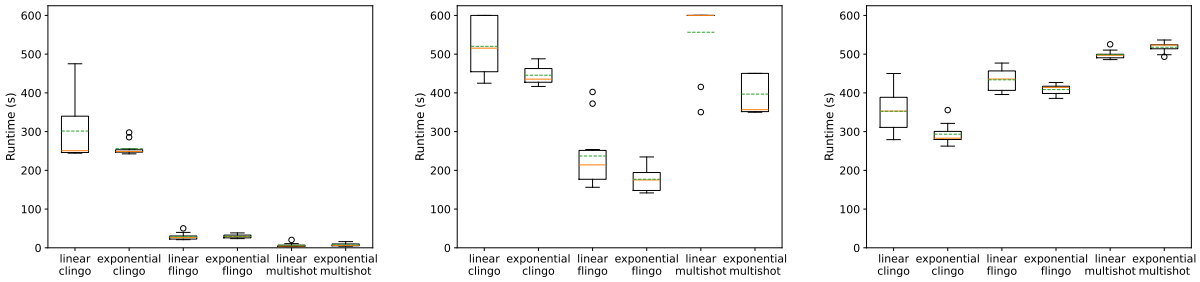
(a) *CargoBike* (b) *TravelBikeFleet* (c) *Restaurant*

Figure 4: Benchmark results for three representative domains

incremental *flingo*, however, it performs better than single-shot and incremental *clingo*. The reason *flingo* performs better here could be due to its dedicated inference algorithm for linear constraints, which enables it to more efficiently solve the combination of minimal volume and maximum price constraints. Lastly, for the *Restaurant*, the multi-shot encoding performs worse than all *clingo* and *flingo* approaches and the *clingo* single-shot approach is able to solve most instances by a margin. For *flingo*, the single-shot and incremental approaches perform comparable but slower than incremental *clingo*.

One possible reason for the worse performance of the multi-shot encoding for instances of the *Restaurant* could be the approach of grounding in steps of 1 and the fact that the minimal upper bounds of its instances reach values of up to 600 (compared to bounds in the range of 5 to 25 for the two bike domains). In general, the multi-shot encoding produces a higher number of ground rules than the single shot encoding due to the representation of incremental constraints. Incremental functions are defined via chains of rules where the length of such a chain equals the bound (see Listing 12). Additionally, for all other types of incremental expressions there are rules defining their value at every possible bound. For the *CargoBike* domain the multi-shot encoding results in around 3 times as many rules for the largest instances. However, for the *Restaurant* domain this factor increases to around 25 due to the higher bounds of the domain. An alternative grounding approach that grounds according to the increments given by the respective algorithm may be advantageous for such domains.

Similarly, the ranges of the minimal upper bounds could explain why smaller linear step sizes perform best on the two bike domains but not for the *Restaurant* (here, for both linear incremental *clingo* and *flingo* a step size of 32 gave the best results). Lastly, note that for the exponential base parameters no clear pattern seems evident, other than a general preference for bases below 2.



(a) *CargoBike* (b) *TravelBikeFleet* (c) *Restaurant*

Figure 5: Average results for linear and exponential search parameters for three domains

Regarding the search algorithms, Figure 4 shows that, generally, the linear and exponential search algorithms perform similarly for their best parameters on all domains. Another question that arises is how much the choice of parameter actually influences the runtime.

Figure 5 shows a boxplot of the average (dashed green line) and median (orange line) runtimes per algorithm and solver for each domain. Note that as a previous step, the average runtimes over all instances had already been calculated and used as basis for the boxplot. Boxes extend from the first

to the third quartile and whiskers extend from the box to the farthest data point lying within 1.5x the inter-quartile range. Outliers lying further outside are represented as small circles.

Generally, the variation of the average runtime for the exponential algorithm is rather small between the evaluated parameters with a minor exception of the *TravelBikeFleet* domain. The linear algorithm deviates more, especially for incremental *clingo* and, most notably, for the linear multi-shot approach on the *TravelBikeFleet*. Here, only for two parameter choices the average runtimes lie below the timeout of 600 seconds (these correspond to the smallest step sizes of 1 and 2; cf. Table 4 in Appendix C). However, there are also many cases with almost no deviation at all. The complete results for each parameter are shown in Appendix C.

5. Discussion

We presented a new and general *clingo* multi-shot encoding for the CoomSuite that works with all language constructs of Coom[x*]. This allows us to solve Coom product models incrementally and offers an alternative to a naive approach, where grounding and solving have to be performed from scratch for each new bound. An evaluation of this approach on three different benchmark domains shows that the multi-shot encoding can increase the performance compared to pure single-shot or naive incremental approaches but that the performance also depends on the structure of the underlying problem. Especially when the minimal upper bounds are large, our current multi-shot encoding performed worse than the other approaches.

Related work Due to their large and intricate nature, the need for solving industrial product configuration problems incrementally arose early on [15]. Other approaches attempt to derive tight upper bounds from UML class diagrams before starting to solve [3] or to automatically infer upper bounds from logical axioms [16].

An ASP multi-shot approach that is similar to ours was introduced in [17] and refined in [18] for the object-oriented configuration language OOASP [19]. However, there, more complex constraints have to be formulated directly in ASP because OOASP does not provide specific constructs for modelling them. While this offers great flexibility, it also obliges the modeller to encode the constraint in the exact format of the given multi-shot encoding.

In contrast, our multi-shot encoding supports all language constructs of fragment Coom[x*]. This follows the principle that the modeller should be able to formulate any knowledge independent of the underlying encoding.

Furthermore, note that we did not evaluate the approach from [3] as they only address cardinalities of UML class diagrams and do not consider more complex dependencies.

Future work Future work consists of extending our benchmark sets and adding refinements to our workflow. For the former, we would like to add language constructs and corresponding encoding parts to model arbitrary associations/connections as is possible for instance in OOASP [19]. This would allow us to evaluate our approach on the the well-known industrial benchmarks from the *Racks* [20] and *House* [21] configuration domains.

Two concrete ideas for refinement of our workflow are implementing different grounding strategies (currently grounding is always done in increments of 1 even when the maximal bound is augmented in larger steps) and the addition of so-called "smart expansion" functions as done in [18]. These functions attempt to infer as much knowledge as possible for each iteration with the aim of guiding the search for minimal bounds. While the functions in [18] were mostly using information from cardinality constraints, the underlying methods should be general enough in order to be applicable to our use case.

Furthermore, instead of just increasing a global bound, we would like to study more fine-grained strategies to increase bounds, which could be useful for incrementally solving problems that are already bounded but have very large cardinalities.

Acknowledgments

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

Declaration on Generative AI

During the preparation of this work, the authors used Gemini and GPT 4.1 in order to: Abstract drafting, Paraphrase and Reword. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

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

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

[3] R. Taupe, A. Falkner, G. Schenner, Deriving tighter component cardinality bounds for product configuration, in: E. Vareilles, L. Hvam, C. Forza, C. Becker (Eds.), Proceedings of the Eighteenth International Configuration Workshop (CONF'16), 2016, pp. 47–54. URL: http://cp2016.a4cp.org/program/workshops/CWS-2016-Proceedings.pdf#section*.8.

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

[5] J. Baumeister, S. Hahn, J. Heuer, K. Herud, M. Ostrowski, J. Reutelshöfer, N. Rühling, T. Schaub, P. Wanko, CoomSuite, https://github.com/potassco/coom-suite, 2024.

[6] J. Baumeister, K. Herud, J. Reutelshöfer, V. Belli, Coom language, https://www.coom-lang.org/, 2024.

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

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

[9] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own ASP-based system?!, Theory and Practice of Logic Programming 23 (2023) 299–361. doi:10.1017/S1471068421000508.

[10] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, P. Wanko, Potassco User Guide, version 2.2.0 ed., 2019. URL: http://potassco.org.

[11] G. Francescutto, K. Schekotihin, M. El-Kholany, Solving a multi-resource partial-ordering flexible variant of the job-shop scheduling problem with hybrid ASP, in: W. Faber, G. Friedrich, M. Gebser, M. Morak (Eds.), Proceedings of the Seventeenth European Conference on Logics in Artificial Intelligence (JELIA'21), volume 12678 of *Lecture Notes in Computer Science*, Springer-Verlag, 2021, pp. 313–328.

[12] N. Rühling, T. Schaub, P. Wanko, Formalizing ASP-based product configuration: Work in progress, in: T. Geibinger, T. Stolzmann, A. Tarzariol (Eds.), Proceedings of the Eigth Workshop on Trends and Applications of Answer Set Programming (TAASP'25), 2025. URL: https://taasp.at/2025/assets/paper/1_Formalizing_ASP_based_Produc.pdf.

[13] P. Cabalar, J. Fandinno, T. Schaub, P. Wanko, flingo — instilling asp expressiveness into linear integer constraints, Theory and Practice of Logic Programming (2026) (to appear).

[14] M. Banbara, B. Kaufmann, M. Ostrowski, T. Schaub, Clingcon: The next generation, Theory and Practice of Logic Programming 17 (2017) 408–461. doi:10.1017/S1471068417000138.

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

[16] M. Aschinger, C. Drescher, G. Gottlob, H. Vollmer, LoCo - A logic for configuration problems, ACM Transactions on Computational Logic 15 (2014) 20:1–20:25. doi:10.1145/2629454.

[17] R. Comploi-Taupe, A. Falkner, S. Hahn, T. Schaub, G. Schenner, Interactive configuration with ASP multi-shot solving, in: J. Horcas, J. Galindo, R. Comploi-Taupe, L. Fuentes (Eds.), Proceedings of the Twenty-fifth International Configuration Workshop (CONF'23), volume 3509, CEUR Workshop Proceedings, 2023, pp. 95–103. URL: https://ceur-ws.org/Vol-3509/paper13.pdf.

[18] L. Balázová, R. Comploi-Taupe, S. Hahn, N. Rühling, G. Schenner, Smart expansion techniques for ASP-based interactive configuration, Theory and Practice of Logic Programming 25 (2025) 473–488. doi:10.1017/S147106842510029X.

[19] 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.

[20] A. Falkner, G. Friedrich, A. Haselböck, G. Schenner, H. Schreiner, Twenty-five years of successful application of constraint technologies at siemens, AI Magazine 37 (2016) 67–80. doi:10.1609/aimag.v37i4.2688.

[21] G. Friedrich, A. Ryabokon, A. Falkner, A. Haselböck, G. Schenner, H. Schreiner, (re)configuration based on model generation, 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. 26–35. doi:10.4204/eptcs.65.3.

A. Converge algorithm for multi-shot encoding

Algorithm 3 shows the converge algorithm for the multi-shot workflow. This corresponds to the function *converge_bound* from Algorithm 1.

The input is a ground program and a lower and upper bound and the output a minimal (satisfiable) bound within the given interval. The given bounds are such that the program is not satisfiable for the lower bound and satisfiable but not (yet) optimal for the upper bound.

As for the general multi-shot algorithm from Algorithm 1, the converge algorithm is almost identical to the procedure described in Section 2.2 for the incremental single-shot workflow. However, the difference is that we need to take care of the corresponding externals such that only the program parts belonging to our current upper bound are active.

B. Benchmark models

Listings 14, 15 and 16 show representative Coom product models of all benchmark domains from Table 1.

Algorithm 3: Converge bound

Input: ground_program
Input: bounds
Output: minimal_bound

while *bounds[0]* $+1 \neq$ *bounds[1]* **do**
    b $\leftarrow \lfloor$(bounds[1] + bounds[0])/2$\rfloor$;
    if *bounds[0]* $< b \leq$ *bounds[1]* **then**
        foreach $b < i \leq$ *bounds[1]* **do**
            Set external `active(i)` to **False**;
    else
        foreach *bounds[1]* $< i \leq b$ **do**
            Set external `active(i)` to **True**;
    Set external `max_bound(bounds[1])` to **False**;
    Set external `max_bound(b)` to **True**;
    SAT $\leftarrow$ solve(ground_program);
    if *SAT* **then**
        bounds $\leftarrow$ (bounds[0],b);
    else
        bounds $\leftarrow$ (b,bounds[1])
minimal_bound $\leftarrow$ bounds[1];

```
product {
            num   0-100   totalVolume
            num   0-100   requestedVolume
    0..*    Bag           bags
            Color         color
}
structure Bag {
            Color       color
            Size        size
    0..*    Pocket      pockets
    1..3    Opening     openings
}
enumeration Pocket { S M L XL }
enumeration Opening { Zipper Velcro }
enumeration Color { Green Blue Red }
enumeration Size {
    attribute num weight
    attribute num volume

    Small   = ( 10 12 )
    Medium  = ( 15 16 )
    Large   = ( 25 20 )
}
behavior {
    require totalVolume = sum(bags.size.volume)
    require totalVolume >= requestedVolume
   require 5 < count(bags.pockets)
}
behavior Bag {
    condition size = Small
    require 2 <= count(pockets)

    condition size = Medium
    require 4 <= count(pockets)

    condition size = Large
    require 6 <= count(pockets)

    combinations (size     pockets)
    allow        (Small    S      )
    allow        (Medium   M      )
    allow        (Large    (L,XL) )

    require 2 <= count(openings)
}
```

Listing 14: Representation of a model of the *CargoBike* benchmark in Coom

```
product {
      num   0-500 maxPrice
      num   0-500 requestedVolume
  0..*  Bike      bikes
}
structure Bike {
      num   0-100 price
      Wheel     frontWheel
      Wheel     rearWheel
      Frame     frame
      Carrier     carrier
}
structure Carrier { 0..3 Bag bag }
structure Frame   { 0..2 Bag bag }
enumeration Bag {
  attribute num   volume
  attribute num price

  B5  = ( 5 3 )
  B10 = ( 10 6 )
  B25 = ( 25 8 )
  B50 = ( 50 12 )
}
enumeration Wheel {
  attribute num  size
  attribute   num  price

  W22 = (  22 5 )
  W24 = (  24 10 )
  W26 = (  26 20 )
  W28 = (  28 25 )
}
behavior {
  require sum(bikes.price) <= maxPrice
  require requestedVolume <= sum(bikes.frame.bag.volume) + sum(bikes.carrier.bag.volume)
}
behavior Bike {
  require count(carrier.bag) + count(frame.bag) <= 2
  require frontWheel.size = rearWheel.size
  imply price = frontWheel.price + rearWheel.price + sum(carrier.bag.price) + sum(frame.bag.price)
}
```

Listing 15: Representation of an unbounded model of the *TravelBikeFleet* benchmark in Coom

```
product {
  0..*  BigTable bigTables
  0..*  SmallTable smallTables
  num 1-700 totalPlaces
}
structure BigTable { 3..5 Chair chairs }
structure SmallTable { 1..2 Chair chairs }
structure Chair {}

behavior {
  imply totalPlaces = count(bigTables.chairs) + count(smallTables.chairs)
}
```

Listing 16: Representation of an unbounded model of the *Restaurant* benchmark in Coom

C. Supplementary benchmark results

In this section, we show the complete results of the evaluation of the `--step` and `--base` parameters for the linear and exponential search algorithms, respectively. For each domain and incremental approach, the average runtime over all instances in seconds is shown. The best performing parameter for each approach is highlighted in bold. Note that the minima are determined before the averages are rounded to whole numbers, thus, it can appear that multiple minima exist, while only a single number is highlighted. In general, the tables show that the best parameter depends on the domain and that within a domain often the best parameters are similar. This indicates that these parameters depend on the underlying structure of the problem as well. For instance, as discussed in Section 4, the minimal bounds of the *Restaurant* domain are typically very large (between 30 and 600) which could explain why higher step sizes for linear search perform better. In contrast, the bounds for the two bike domains are in the range of 5 to 25 and thus smaller step sizes seem to be more effective.

Table 2
Average runtimes (in seconds) for `--step` parameters of linear search algorithm for clingo

	1	2	3	4	6	8	12	16	24	32
CargoBikePockets	249	246	245	253	**244**	246	252	369	434	475
Restaurant	450	410	391	381	363	343	311	311	286	**279**
TravelBikeFleet	**425**	432	452	462	487	544	600	600	600	600

Table 3
Average runtimes (in seconds) for `--step` parameters of linear search algorithm for flingo

	1	2	3	4	6	8	12	16	24	32
CargoBikePockets	29	24	**21**	23	22	22	28	31	40	50
Restaurant	477	461	459	450	443	429	422	401	400	**396**
TravelBikeFleet	157	**156**	217	176	211	180	244	372	254	402

Table 4
Average runtimes (in seconds) for `--step` parameters of linear search algorithm for multishot

	1	2	3	4	6	8	12	16	24	32
CargoBikePockets	**3**	3	3	4	5	5	6	7	11	20
Restaurant	497	500	496	489	487	**486**	497	510	525	499
TravelBikeFleet	**350**	415	600	600	600	600	600	600	600	600

Table 5
Average runtimes (in seconds) for `--base` parameters of exponential search algorithm for clingo

	1.25	1.33	1.50	1.66	1.75	2.00	2.25	2.33	2.50	2.66	2.75
CargoBikePockets	249	255	244	247	**243**	247	251	249	247	285	298
Restaurant	311	281	**263**	356	282	276	278	286	290	321	283
TravelBikeFleet	**417**	424	432	418	431	435	446	449	485	488	477

Table 6
Average runtimes (in seconds) for `--base` parameters of exponential search algorithm for flingo

	1.25	1.33	1.50	1.66	1.75	2.00	2.25	2.33	2.50	2.66	2.75
CargoBikePockets	26	**24**	24	25	29	28	30	32	33	38	37
Restaurant	419	388	**386**	414	414	427	398	415	427	399	406
TravelBikeFleet	149	148	**142**	169	143	175	217	235	202	186	181

Table 7
Average runtimes (in seconds) for `--base` parameters of exponential search algorithm for multishot

	1.25	1.33	1.50	1.66	1.75	2.00	2.25	2.33	2.50	2.66	2.75
CargoBikePockets	**4**	5	5	6	5	6	6	8	12	14	16
Restaurant	**493**	515	499	525	512	524	524	518	526	537	524
TravelBikeFleet	352	**350**	352	351	351	357	450	450	450	450	450