TPLP **26** (3-4): 340–370, 2026. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068426100404 First published online 1 April 2026

*Towards Industrial-Scale Product Configuration**

JOACHIM BAUMEISTER
denkbares GmbH, Germany and University of Würzburg, Germany
(*e-mail:* joba@uni-wuerzburg.de)

SUSANA HAHN
Institute of Computer Science, University of Potsdam, Germany and Potassco Solutions GmbH, Germany
(*e-mail:* hahnmartinlu@uni-potsdam.de)

KONSTANTIN HERUD
denkbares GmbH, Germany
(*e-mail:* konstantin.herud@denkbares.com)

MAX OSTROWSKI
Potassco Solutions GmbH, Germany
(*e-mail:* ostrowsk@cs.uni-potsdam.de)

JOCHEN REUTELSHÖFER
denkbares GmbH, Germany
(*e-mail:* jochen.reutelshoefer@denkbares.com)

NICOLAS RÜHLING
Institute of Computer Science, University of Potsdam, Germany and UP Transfer GmbH Germany,
(*e-mail:* nruehling@uni-potsdam.de)

TORSTEN SCHAUB
Institute of Computer Science, University of Potsdam, Germany and Potassco Solutions GmbH, Germany
(*e-mail:* torsten@cs.uni-potsdam.de)

PHILIPP WANKO
Potassco Solutions GmbH, Germany
(*e-mail:* wanko@cs.uni-potsdam.de)

submitted 20 March 2025; revised 17 October 2025; accepted 20 January 2026

* This work was partly funded by ZIM BMBK grants KK5291307GR4 and KK5394902GR4. We, would like to thank Simon Vandevelde and the anonymous reviewers for their valuable feedback.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

Abstract

We address the challenge of product configuration in the context of increasing customer demand for diverse and complex products. We propose a solution through a curated selection of product model benchmarks formulated in the Coom language, divided into three fragments of increasing complexity. Each fragment is accompanied by a corresponding example on bike configuration, and additional scalable product models are included in the CoomSuite, along with relevant resources. We outline an ASP-based workflow for solving Coom-based configuration problems, highlighting its adaptability to different paradigms and alternative ASP solutions. The CoomSuite aims to provide a comprehensive, accessible, and representative set of examples that can serve as a common ground for stakeholders in the field of product configuration.

KEYWORDS: product configuration, answer set programming, Coom, interactive configuration

1. Introduction

Configuration (Felfernig *et al.* 2014), a longstanding challenge in AI, is gaining renewed attention as manufacturing industries grapple with the challenge of meeting customer demands for high variance and complexity, all while operating under a high level of automation. To achieve this, configuration processes must empower customers to tailor products to their specific needs within a predefined configuration space. This space is designed to seamlessly integrate with subsequent processes (pricing, quotation, manufacturing, resource planning, and delivery), enabling them to be executed in a highly automated manner. Despite the existence of various approaches to product configuration (Felfernig *et al.* 2014), building and utilizing complex product models for this purpose remains a challenge in industrial settings due to the diversity of methods, representations, and configuration systems available. Furthermore, stakeholders from industry, research, and software development often struggle to align their understanding of the practical application task with the precise semantics of the modeling formalism.

To address this challenge, we propose a curated selection of product model benchmarks that encapsulate the key challenges of product configuration. These benchmarks can serve as a common ground for fostering a meaningful exchange among all stakeholders involved. More precisely, we formulate our benchmarks in the industrial configuration language Coom (Baumeister *et al.* 2025b),[1] and distinguish three language fragments of increasing complexity. The basic one, CoomCore, is mainly about (discrete) attributes and resembles a simple constraint satisfaction problem. The extended one, Coom[p], adds partonomies and cardinalities. And the last considered Coom fragment, Coom[x] adds numeric variables and calculations on top of Coom[p]. Each such class is accompanied by an exemplary product model from the domain of bikes, more precisely, a *KidsBike*, *CityBike*, and *TravelBike* model in Coom. Extensions for two of these language fragments called Coom[p*] and Coom[x*], respectively, enable modeling with unbounded cardinalities, where the latter is exemplified by the product model of a *CargoBike*. These and further scalable product models are bundled together in the CoomSuite (Baumeister *et al.*

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

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

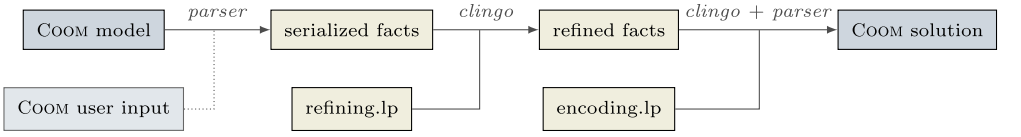


Fig 1. Workflow for solving Coom configuration problems with ASP.

2025a), which is intended to serve as a workbench for experimentation with industrial-scale product configuration problems and, in addition to all benchmark files, contains Coom grammar definitions and further resources related to Answer Set Programming (ASP; Lifschitz, 2019).

The workflow for solving Coom-based configuration problems with ASP is outlined in Figure 1. We start by converting product models in the form of a Coom specification and, optionally, a user input into ASP facts. This conversion is guided by a Coom grammar, which is supplied through the CoomSuite. A specialized ASP visitor then processes the resulting parse tree, translating it into a corresponding set of facts. This initial set of facts serves as a direct serialization of the tree structure and is further refined through an ASP encoding tailored to the specific ASP solution being implemented. The refined fact format, in conjunction with its corresponding configuration encoding, comprehensively defines the configuration space. Ultimately, the answer sets representing valid configurations are transformed back into Coom solutions. Additionally, the CoomSuite offers the possibility to interactively solve configuration problems through a Coom-specific but application-independent user interface (UI).

While this workflow is primarily geared towards ASP, it is adaptable in several ways. First, the serialization of Coom specifications can be tailored for different paradigms by utilizing a specialized parse tree visitor. Second, alternative ASP solutions can be investigated by modifying the refinement of the fact format and employing a different configuration encoding.

Our goals in developing the CoomSuite are to provide and evaluate a comprehensive set of product model examples that are:

Accessible Easily understandable for individuals with diverse backgrounds, including researchers and product managers.
Representative Encompass the typical challenges encountered in product configuration over recent decades.
Unifying Serve as a shared reference point for discussions on representations and formalisms in the field.
Facilitative Aid in the development, testing, and comparison of software components designed for product configuration.
Open Freely available for public use and contribution.

To the best of our knowledge, there currently exists no publicly accessible set of product configuration examples that meets our criteria.

The remainder of our paper is organized as follows: Section 2 introduces the CoomCore language and demonstrates its application through the *KidsBike* example. Section 3

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

presents our ASP-based approach to product configuration, detailing the refined fact format and ASP encoding. Section 4 outlines various extensions to the CoomCore language which contain commonly used features in product modeling and how they are handled within our ASP framework. These include concepts such as partonomy, cardinalities, and numerics, as well as user input and unbounded cardinalities. Further, an alternative encoding for solver *flingo* (Cabalar *et al.* 2026) is presented. Finally, Section 5 describes the CoomSuite UI along with its ASP implementation through system *clinguin,* and Section 6 demonstrates the practical value of the CoomSuite by evaluating alternative ASP solutions against benchmarks of varying complexity levels.

Related work

Early on, a general ontology for representing product configuration problems was introduced in (Soininen *et al.* 1998), which inspired many of Coom's concepts. Coom also draws inspiration from other ontologies aiming to represent corporate knowledge in a standardized way (Brunner *et al.* 2007; Coskun *et al.* 2009), especially in the context of products and industrial manufacturing (Lin *et al.* 1997; Baumeister 2018). A wide range of approaches exist for representing and solving configuration problems across various paradigms (Junker 2006; Hotz *et al.* 2014). In recent years, ASP has emerged as a promising alternative, as evidenced by several applications (Gebser *et al.* 2011; Felfernig *et al.*, 2017; Gençay *et al.* 2019; Herud *et al.* 2022). Moreover, Falkner *et al.* (2015) developed an object-oriented approach to configuration by directly defining concepts in ASP, and Rühling *et al.* (2023) recently proposed an ASP-based approach rooted in firm mathematical foundations. In the context of interactive configuration, Falkner *et al.* (2020) conducted a comparative evaluation of various systems, including the ASP solver *clingo* as well as SAT and CP systems, for their suitability in this context, finding *clingo* to be as capable as any other system.

Another system for interactive configuration has been developed by Carbonnelle *et al. (*2023), which uses the IDP-Z3 reasoning engine to interactively solve configuration problems specified in the FO(·) language. This has been successfully applied to various applications such as finding suitable adhesives (Vandevelde *et al.* 2024) and machine component design (Aerts *et al.* 2022).

Furthermore, in the related context of product-line variability modeling and configuration, the Clafer language (Juodisius *et al.* 2019) has been developed, which allows to capture in a unified way feature models (Felfernig *et al.* 2024), component models, and others. A series of supporting tools (Antkiewicz *et al.* 2013a) are available of which we highlight the interactive, web-based Clafer Configurator (Antkiewicz *et al.* 2013a,b).

2. The CoomCore language

In this section, we introduce the CoomCore language fragment through the illustrative *KidsBike* example. This fragment primarily corresponds to basic constraint satisfaction problems (Dechter 2003), focusing on variables that need to satisfy a given set of constraints. The *KidsBike* example, presented in Listing 1, encapsulates a simplified product configuration problem and incorporates all the essential language features of CoomCore.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

```
1  product {
2     Color    color
3     Bool     wheelSupport
4     Wheel    frontWheel
5     Wheel    rearWheel
6  }
7  enumeration Color { Red Green Yellow Blue }

9  enumeration Wheel {
10    attribute  num  size
11    attribute  num  price

13    W14 = ( 14 50 )    W16 = ( 16 60 )
14    W18 = ( 18 70 )    W20 = ( 20 80 )
15 }
16 behavior {
17    condition color = Yellow require frontWheel.size > 16

19    combinations  (wheelSupport  rearWheel)
20    allow         (True          (W14, W16))
21    allow         (False         (W18, W20))

23    require frontWheel.size = rearWheel.size
24 }
```

Lisiting 1. Representation of the *KidsBike* example in the Coom language

Line 1 establishes the definition of a `product`. This represents the fundamental entity of the object we intend to configure. Note that only one product can be defined within a single Coom file. Inside a `product` several *features* are declared (Lines 2–5). A feature is declared by stating its type followed by its name, as in `"Color color"`. In this case, type and name are differentiated solely by capitalization. However, for the type `Wheel`, we have two distinct names: `frontWheel` and `rearWheel`. The type `Bool` is a built-in type in Coom. In CoomCore, a type of a feature must refer to an *enumeration*, representing finite domains of predefined choices. A simple example is the `Color` enumeration in Line 7. Furthermore, enumerations can include *attributes* to assign specific values to each option. For instance, the `Wheel` type defines two attributes: `size` and `price` (Line 10 and 11), and its options are declared in Lines 13–14. Unlike the `Color` enumeration, each `Wheel` option includes additional values for the `size` and `price` attributes, enclosed within parentheses. This indicates, for example, that wheel `W16` has a `size` of `16` and a `price` of `60`.

Features in CoomCore can be seen as variables and thus be used for expressing constraints over them. Coom also offers to use attributes of enumerations as variables, for example, by writing `frontWheel.size` to refer to the `size` attribute of the feature `frontWheel`. In general, these variables are called *path expressions*.

Constraints can be declared within `behavior` structures. Here, we allow for three types of constraints. In Line 17, a *conditional requirement* is declared, stating that the `size` of the front wheel has to be greater than `16` if the `color` of the bike is `Yellow`. Such constraints are declared by using the keywords `condition` and `require`, followed by a

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

simple logical (comparison) statement. These statements can contain path expressions and constants, numbers, or strings. In Line 23, a condition-free requirement is defined, stating that the size of the front and rear wheel have to be equal.

Another type of constraint is implemented through *combination tables*. In the example, Lines 19–21 specify that the use of a `wheelSupport` is obligatory with small wheels. More specifically, this constraint is declared using the keyword `combinations` in Line 19, followed by a list of path expressions as the table's column headers. Then, Lines 20 and 21 define the allowed combinations. A table entry can contain multiple values, as illustrated in the `rearWheel` column. Any combination that aligns with an `allow` line within the table represents a valid configuration for the product, while those that do not are invalid.

3. Solving Coom-based product configuration with ASP

We now present our ASP-based approach to solving product configuration problems defined in CoomCore. As previously mentioned, we begin by converting a Coom input file into a set of facts that represent the serialized parse tree of the Coom model. This is accomplished by using a custom Python *ANTLR v4* parser, which utilizes a Coom grammar, making it easily adaptable to future language upgrades. Next, we transform[2] the initial set of syntax-reflecting facts into a more refined representation that captures the essential concepts of the configuration problem in view of the configuration encoding at hand. This abstraction allows us to move beyond the specific details of the input format and focus on modeling the fundamental concepts of the configuration problem.

More precisely, the basic concepts of the refined fact format are variables whose type is a discrete attribute over a finite domain. There are two types of constraints over these variables: Boolean and table constraints. For illustration, we present (an extract of) the refined fact format of the *KidsBike* in Listing 2. Our fact format uses an object-centered representation, which identifies all concepts with terms.

Variables are identified by strings such as "`color`" or "`frontWheel`" and their types are declared in Lines 1–3.

Lines 5–7 give the (discrete) attributes of the configuration model. In fact, both the `Wheel` enumeration and one of its enumeration attributes `size` from the Coom model are represented as discrete attributes. The `Bool` attribute is a Coom built-in and added automatically during refinement. We omit showing the `Color` and `price` attribute. In Coom, the (combined) values of such attributes are among the enumeration's options (cf. Section 2). We guarantee their compatibility, for example, of `size` and `price` from the `Wheel` enumeration, by means of a table constraint which is omitted for brevity.

Lines 9–20 give three constraints of the configuration model; they reflect Coom `behavior`[s]. Each constraint is identified with a term, `c0`, `c1`, etc., during refinement. Constraint `c0` is a conditional requirement stating that the color `Yellow` is only available with larger wheels. For this, we require in Lines 19–20 that `color=Yellow` is false or `frontWheel.size>16` is true. These statements are represented by `f0` and `f1`, respectively, and the operators `"||"` and `"!"` stand for disjunction and negation,

[2] We omit this syntactic translation since it provides no further insights.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

```
1  type("color","Color").        type("wheelSupport","Bool").
2  type("frontWheel","Wheel"). type("frontWheel.size","size").
3  type("rearWheel","Wheel").   type("rearWheel.size","size").

5  discrete("Wheel").   domain("Wheel",("W14";"W16";"W18";"W20")).
6  discrete("size").    domain("size",(14;16;18;20)).
7  discrete("Bool").    domain("Bool",("True";"False")).

9  constraint(c0,"boolean").
10 binary(c0,f0,"||",f1).                    unary(f0,"!",f2).
11 binary(f2,"color","=","Yellow").          constant("Yellow").
12 binary(f1,"frontWheel.size",">","16"). number("16",16).

14 constraint(c1,"table").
15 column(c1,0,"wheelSupport"). column(c1,1,"rearWheel").
16 allow(c1,(0,0),"True").       allow(c1,(1,0),("W14";"W16")).
17 allow(c1,(0,1),"False").      allow(c1,(1,1),("W18";"W20")).

19 constraint(c2,"boolean").
20 binary(c2,"frontWheel.size","=","rearWheel.size").
```

Lisiting 2. Extract of the refined ASP representation of the *KidsBike* example

respectively. They are further decomposed via predicates `binary/4` and `unary/3` until the variable or constant level is reached. Analogously, Lines 14–17 encode a Boolean constraint stating that the size of the front and rear wheel must be equal.

Lastly, Lines 14–17 give a table constraint reflecting a `combinations` table in Coom. The constraint and its columns are declared in Lines 14 and 15, followed by the table entries in Lines 16 and 17, expressed by predicate `allow/3`. Note that multiple values can be given for a table entry, as for entries `(1,0)` and `(1,1)`.

In what follows, we present a simple configuration encoding for CoomCore models in Listing 3. While Line 1 generates exactly one value per discrete attribute from an associated domain, Line 2 generates auxiliary values for constants used in the constraints.

Next, Lines 4–17 encode Boolean and table constraints. To start with, the integrity constraint in Line 4 makes sure that all original constraints are satisfied. Lines 6–13 specify satisfaction conditions for various types of binary and unary formulas making up a Boolean constraint, viz. disjunction (Lines 6 and 7), negation (Line 8), and comparison operators = and > (Lines 10–13). To encode disjunction, we make use of *clingo*'s weight constraints, where the expression `1 <= satisfied(XL); satisfied(XR)` states that at least one of the two subformulas `XL` or `XR` must be satisfied.

Lines 15–17 give satisfaction conditions for table constraints. For this, we identify all rows that are not satisfied (Lines 15 and 16). A row is not satisfied whenever one of its entries is not satisfied. Note that as entries can contain multiple values, we employ *clingo*'s conditional literals to check for all of the individual values within the table entry. Here, the expression `not value(X,V) : allow(C,(Col,Row),V)` will be expanded to `not value(X,V1), not value(X,V2),...` during grounding for a table entry containing the tuple `(V1,V2,...)`. More precisely, the expression `not value(X,V)`

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

```
1  { value(X,V) : domain(T,V) } = 1 :- type(X,T), discrete(T).
2  value(P,P) :- constant(P). value(P,N) :- number(P,N).

4  :- constraint(C,_), not satisfied(C).

6  satisfied(F) :- binary(F,XL,"//",XR),
7                  1 <= { satisfied(XL); satisfied(XR) }.
8  satisfied(F) :- unary(F,"!",F'), not satisfied(F').

10 satisfied(F) :- binary(F,XL,"=",XR), VL = VR,
11                 value(XL,VL), value(XR,VR).
12 satisfied(F) :- binary(F,XL,">",XR), VL > VR,
13                 value(XL,VL), value(XR,VR).

15 nhit(C,Row)  :- allow(C,(Col,Row),_), column(C,Col,X),
16                 not value(X,V) : allow(C,(Col,Row),V).
17 satisfied(C) :- allow(C,(_,Row),_), not nhit(C,Row).

19 #show value/2.
```

Listing 3. ASP encoding of refined CoomCore

`: allow(C,(Col,Row),V)` gathers all possible values `V1,V2,...` for a given table entry `(Col,Row)` of the table constraint `C` and then expands to a conjunction of literals `not value(X,V1), not value(X,V2),...`. Finally, a table constraint is satisfied if one of its rows is satisfied (Line 17).

Listing 4 shows how to solve the *KidsBike* example with the `solve` mode of the CoomSuite following the workflow from Figure 1. Currently, the CoomSuite uses *clingo* as the default solver. Here, `kids-bike.coom` is the input file containing the Coom model of the *KidsBike* example corresponding to Listing 1. The option `--output coom` (or `-o coom` for short) converts the ASP output to a (more readable) Coom format.

```
$ coomsuite solve kids-bike.coom --output coom
COOM Suite version 0.1
Reading from kids-bike.lp
Solving...
Answer: 1
color[0] = "Red"
wheelSupport[0] = "True"
frontWheel[0] = "W14"   frontWheel[0].size[0] = 14
rearWheel[0]  = "W14"   rearWheel[0].size[0]  = 14
SATISFIABLE
```

Listing 4. Solving the *KidsBike* example with the CoomSuite

4. Coom language extensions

Having explained the basics of Coom, we now proceed to elaborate on its extensions and how they can be modeled and solved using ASP within our framework.

We begin by showing how more commonly used features like partonomy, cardinalities, and numeric calculations can be modeled within Coom[x], using our *TravelBike* example

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

for illustration. We do not delve into Coom[p], as all its concepts are already encompassed within Coom[x].

We start by explaining how these features are implemented and how they are used in the *TravelBike* example. The bike's partonomy consists of two parts: a carrier and a frame. Both of them can be equipped with zero, one, or multiple bags, thus making bags an optional component. However, the model's constraints can necessitate a minimum total storage volume greater than zero, effectively requiring the inclusion of at least one bag. To calculate the total storage volume, arithmetic aggregation functions are employed to sum the storage capacity across all bags.

Similar to above, we first outline how these features are modeled within Coom[x] and then demonstrate their representation and resolution using ASP. For this part, we first present the fact format and then proceed to show two similar but alternative encodings: one for solver *clingo* and one for *flingo*. For both, we showcase selected aspects and refer the reader to Baumeister *et al.* (2025a) for the complete encoding.

In the second part of this section, we use Coom[x] as a basis to describe two less commonly used features which are nevertheless equally important in practice. The first one is so-called user input which allows for the specification of custom requirements at runtime such as for example a minimum total storage volume for the abovementioned *TravelBike*, thereby excluding certain solutions from the search space beforehand.

The second feature are unbounded cardinalities, which allow modeling cases where the exact (or approximate) number of some object is not known beforehand. Here, we introduce extensions to two of our language fragments, which we call Coom[p*] and Coom[x*], respectively. We illustrate Coom[x*] using the *CargoBike* example, a modified version of the *TravelBike*, where the number of possible bags is unbounded. For both, we proceed in the same manner as before by first showcasing a Coom example and then showing how this is solved using ASP.

4.1 The TravelBike in Coom[x]

Listing 5 gives a simplified Coom[x] representation of the *TravelBike* example. First, the features of the bike are defined, starting with the two numeric ones, `totalVolume` and `requestedVolume` in Lines 2 and 3. Both are marked as such by the keyword `num` at the beginning of the line, followed by their respective ranges. The second feature can be thought of as a user requirement that can be set at runtime (cf. Section 4.4). Next, Lines 4 and 5 define the `carrier` and `frame`, whose types are captured by `structure`[s] in Lines 7 and 8. Both the `num` keyword and the `structure` keyword were not possible before in CoomCore. Other than `enumeration`[s], `structure`[s] may have features on their own, which allows for building complex partonomies. For example, the `Carrier` and the `Frame` structure have exactly one feature, `bag`. The expressions `0..3` and `0..2` give their respective cardinalities. Strictly speaking, each feature in Coom has a cardinality, but when omitted it defaults to `1..1`. In all our Coom fragments until now, lower and upper bounds are mandatory, and we carry this requirement over to our fact format as well (only in Section 4.5 we generalize this to allow for open bounds.) The type of both features is `Bag`, which in turn is an `enumeration` with a single attribute `volume` (Lines 10–13).

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

```
1  product {
2      num 0-200 totalVolume
3      num 0-200 requestedVolume
4      Carrier   carrier
5      Frame     frame
6  }
7  structure Carrier {0..3 Bag bag}
8  structure Frame   {0..2 Bag bag}

10 enumeration Bag {
11     attribute num volume
12     B20  = ( 20 )   B50  = ( 50 )   B100 = ( 100 )
13 }
14 behavior {
15     require count(carrier.bag) + count(frame.bag) <= 4

17     require totalVolume = sum(carrier.bag.volume) +
18                           sum(frame.bag.volume)

20     require totalVolume >= requestedVolume
21 }
```

Listing 5. Simplified representation of the *TravelBike* in Coom

Lastly, we discuss the constraints of the model. The first constraint requires that there are no more than four bags in a configuration (Line 15). For this, it uses aggregate functions to `count` the number of bags. In general, aggregate functions perform calculations over a set of variables, which are defined implicitly in terms of a path expression, for example `carrier.bag`. In this case, the function `count(carrier.bag)` returns the actual number of bags attached to the `carrier` in the configuration at hand.

We introduced path expressions in Section 2 as variables corresponding to a product feature, or an enumeration attribute, respectively. This is valid for CoomCore, however, for Coom[p] and Coom[x] we need a more general definition as features can have cardinalities different than 1. We now say that a path expression serves as an identifier for a set of variables.[3] Each part of the path expression is a feature name and the last part can be an enumeration attribute.

Another aggregate function is `sum`, which returns the sum of the values of all variables in a set (Lines 17 and 18). Here, the constraint requires that the value of `totalVolume` is equal to the sum of the volume of all bags. The last constraint is another `requirement` and relates the calculated value of `totalVolume` to the value `requestedVolume`.

4.2 Solving Coom[x] in ASP

In this section, we outline our ASP-based approach to solving product configuration problems specified in Coom[x]. We start by illustrating the refined fact format representation using the *TravelBike* example, which incorporates more advanced concepts than the fact format presented in Section 3.

[3] In CoomCore this reduces to the case of singleton sets with cardinality 1.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

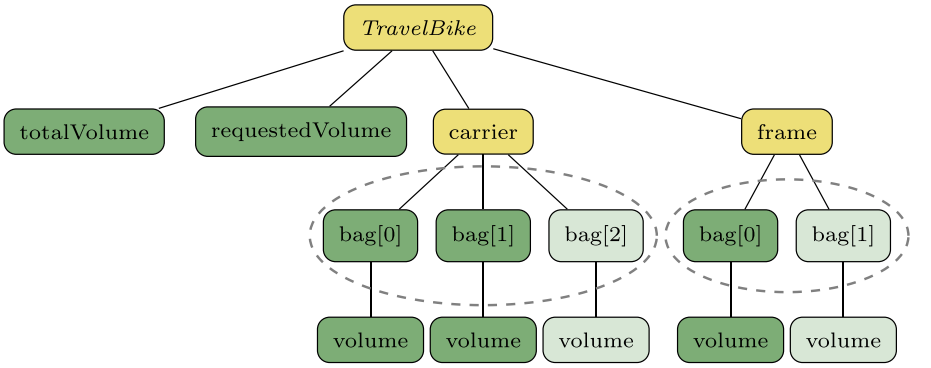


Fig 2. The *TravelBike* example converted into a configuration tree.

The essential concepts of this new fact format are a partonomy with bounded cardinalities, attributes ranging over discrete or integer domains, and Boolean as well as table constraints where the former now allow for arithmetic expressions and aggregate functions.

We represent the configuration as a tree such that each node represents a variable, and its root reflects the object to be configured. Variables can be either parts or attributes. Solutions are represented as subtrees of this configuration 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 (possible) subtree excluded. Cardinalities are represented as constraints over sets of nodes in the tree.

4.2.1 Fact format

In Coom, a partonomy is established by defining structures and linking them through features. These features can also reference enumerations or numeric variables, potentially with cardinalities (as illustrated in Listing 5). We simplify this approach by treating both part and attribute variables uniformly and refer to the resulting structure as the *configuration tree* which can be considered an explicit representation of the Coom configuration product model from before.

For the *TravelBike*, this structure, which serves as the basis for our fact format, is visualized in Figure 2. Here, nodes belonging to parts are highlighted in yellow and those belonging to attributes in green. Consider an example solution where the third bag of the carrier and the second bag of the frame are 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. Note that for the sake of readability variable names are abbreviated and value assignments are omitted. Cardinalities of features are treated by grouping variables belonging to the same feature and with the same parent variable in sets (represented by dashed circles in the diagram).

Our refinement process generates all possible variables within the model and assigns them unique (human-readable) identifiers by leveraging the tree-like structure of the configuration. For example, the `frame` feature of the `product` in Listing 5 results in the

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

variable `"root.frame[0]"`. In this context, `"root"` refers to the object being configured, in our case, the *TravelBike*. Indices are required as feature cardinalities are potentially larger than 1. For instance, the `bag` feature of the `Frame structure` yields two variables `"root.frame[0].bag[0]"` and `"root.frame[0].bag[1]"` which can be read as "the first and second bag of the first frame", respectively. For capturing each such variable in an object-centered representation, we include three facts representing the type, index and parent of the variable.

```
1  type("root.frame[0].bag[1]","Bag").
2  index("root.frame[0].bag[1]",1).
3  parent("root.frame[0].bag[1]","root.frame[0]").
```

Listing 6. A variable of the instantiated *TravelBike*

We represent Coom's `structure`[s] as parts. Hence, we encode the *TravelBike*'s parts as `part("Carrier").` and `part("Frame").` Since `Bag` is an `enumeration` in Coom, it is represented as a discrete attribute (cf. Section 3).

Cardinalities are represented as lower bound constraints. The upper bound is compiled away during our refinement process by generating the corresponding number of variables. For example, the cardinalities for the bags of the frame are encoded as follows:

```
1  constraint(("root.frame[0].bag",0),"lowerbound").
2  set("root.frame[0].bag","root.frame[0].bag[0]").
3  set("root.frame[0].bag","root.frame[0].bag[1]").
```

Listing 7. Cardinality constraint for the bags of the *TravelBike* frame

While the second argument `"lowerbound"` of predicate `constraint/2` marks its type, the first one contains a pair consisting of an identifier `"root.frame[0].bag"` and the actual lower bound 0. The identifier acts as a representative for a set of variables. This set is encoded by means of predicate `set/2`, whose first argument identifies the set and its second one a set member. The upper bound 2 is thus reflected by the number of set elements.

A numeric attribute and its corresponding range is declared as follows:

```
1  integer("totalVolume").
2  range("totalVolume",0,200).
```

Listing 8. Integer attribute `totalVolume` of the *TravelBike*

As before, a variable is associated with this attribute through predicate `type/2`, for example `type("root.totalVolume[0]","totalVolume")`. Currently, the only numeric attribute type is *integer*.

Aggregate functions are represented in analogy to cardinalities in a set-based fashion.

```
1  function("count(root.frame.bag)","count","root.frame.bag").
2  set("root.frame.bag","root.frame[0].bag[0]").
3  set("root.frame.bag","root.frame[0].bag[1]").
```

Listing 9. Count aggregate function of the *TravelBike*

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

The predicate `function/3` comprises the function's identifier, its type (e.g., `count`), and a set identifier. As above, the members of the set are declared via `set/2`.

Lastly, (binary) arithmetic functions like + or − are represented in the same manner as Boolean binary functions (cf. Listing 2).

4.2.2 Encoding

We now describe selected aspects of our encoding for solving configuration problems in Coom[x]. We start by describing the rules related to the configuration tree.

```
1 { include(X) : type(X,_) }. include("root").

3 :- include(X), parent(X,P), not include(P).

5 :-  include(X),  set(P,X ), index(X, I), I > 0,
6 not include(X'), set(P,X'), index(X',I-1).

8 :- constraint((P,L),"lowerbound"), set(P,X), index(X,L-1),
9    parent(X,X'), include(X'), not include(X).
```

Listing 10. Definedness of variables in Coom[x]

Line 1 of Listing 10 generates instances of `include/1` predicates, stating that a variable of the configuration tree is included in the solution. This rule applies to part as well as attribute variables; the root variable is always included. Line 3 restricts this to variables whose parent variables are included as well. For symmetry breaking, Lines 5 and 6 enforce that a variable with index `I` is only included if the variable with index `I-1` from the same (cardinality) set is included as well. We leverage this for enforcing cardinality constraints in Lines 8 and 9. Since variables in a cardinality set are included in ascending index order, we only need to check that the variable whose index corresponds to the lower bound is included in the solution. Also, a cardinality is only enforced if the parent variable is included in the solution.

Next, we consider numerics in Listing 11 for generating attribute values.

```
1 { value(X,V) : domain(T,V) } = 1 :- include(X), type(X,T),
2                                     discrete(T).
3 { value(X,V) : V = L..U    } = 1 :- include(X), type(X,T),
4                                     integer(T), range(T,L,U).
```

Listing 11. Generation of attribute values in Coom[x]

Unlike Section 3, we now need to take into account possibly undefined (attribute) variables. In our current encoding, we say that an attribute variables is defined iff it is included in the solution. The rule in Lines 1 and 2 is similar to the one above, just that now the body additionally contains the atom `include(X)`. This guarantees that an attribute variable only generates a value if it is included in the solution and thus not undefined. Integer attributes are handled analogously except that values are generated from a range of numbers.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

The possible undefinedness of variables also has an effect on constraint satisfaction and violation. For example, in our CoomCore-encoding, a constraint is violated when it is not satisfied. This condition is no longer sufficient here, as variables in a constraint can now be undefined. We therefore need to adapt the integrity constraints accordingly.

```
1  :- constraint((C,F),"boolean"), defined(F), not satisfied(F).
2  defined(X) :- value(X,_).
3  defined(F) :- binary(F,XL,_,XR), defined(XL), defined(XR).
```

Listing 12. Boolean constraint checking in Coom[x]

The integrity constraint in Listing 12 contains the additional atom `defined(F)`, making sure that the Boolean formula `F` can only be violated if it is actually defined. The predicate `defined/1` is specified recursively. An (attribute) variable is defined when it has an assigned value and a binary predicate is defined when both of its parts are defined.

Lastly, we showcase the encoding of arithmetic and aggregate functions.

```
1  value(F,V) :- function(F,"count",P),
2                V = #count{ X    : set(P,X), include(X) }.
3  value(F,V) :- function(F,"sum",  P),
4                V = #sum  { V',X : set(P,X), value(X,V')}.
5  value(F,VL+VR) :- binary(F,XL,"+",XR),
6                    value(XL,VL), value(XR,VR).
```

Listing 13. Numerical calculations in Coom[x]

Listing 13 contains rules for the `count` and `sum` aggregate function as well as for the + operator. For all three rules we are making use of built-in *clingo* functionality. Note that whenever non-domain predicates (`include/1` and `value/2`) occur inside an aggregate, assigning the result of that calculation to another variable possibly creates very large groundings. This is due to the fact that all possible value assignments have to be explicitly represented. However, in this encoding we opted for clarity and simplicity over performance and leave optimized versions for future work.

Solving Coom[x] product models with the CoomSuite works by using its `solve` mode in the same fashion as already shown in Listing 4 above for the *KidsBike* example. We do not show the command-line in- and output here but defer this to Section 4.4 where we introduce user input.

4.3 Solving Coom[x] with flingo

An alternative Coom[x] encoding is given for solver *flingo*[4] which is a prototype system for solving conditional linear constraints with integer variables in ASP. This system is a continuation of the *lc2casp* system (Cabalar *et al.* 2016) enhanced with conditional linear constraints given via the translation in (Cabalar *et al.* 2020). *flingo* uses CASP

[4] *fclingo* in earlier versions.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

solver *clingcon* (Banbara *et al.* 2017) as a backend, therefore being 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 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 from the previous section, except for rules which treat integer attributes and numerical constraints.

We start by giving a brief overview of the *flingo* features we are utilizing for this adaptation and then proceed to highlight the differences between the two encodings. Our main objective with the *flingo* encoding is to improve performance for numerical instances by overcoming *clingo*'s grounding bottleneck. This is achieved in *flingo* by allowing for a special type of variables which are not subject to grounding. These are called *integer variables* and as their name suggests, can take values from the domain of the integers. Importantly, they are being treated differently than propositional variables. Further, to be able to properly replace the necessary rules from our earlier encoding, we require that these integer variables can be used to build linear constraints while comprising a notion of undefinedness and the possibility to define defaults. Lastly, we also need to be able to exclude variables from a calculation if they do not fulfill certain conditions, that is aggregate functions should allow for conditionality. *flingo* provides us with all these features as we see in the following. Rules in *flingo* are written in the same way as in *clingo* but additionally there are a few special theory atoms through which we can define integer variables and build up constraints over them. Here, we explain only those theory atoms that we are using in our encoding and refer the reader to Cabalar *et al.* (2025), 2026) for further documentation and examples. We proceed by showing snippets of the *flingo* encoding while highlighting the differences from the *clingo* encoding from Section 4.2.2.

First, Line 1 in Listing 14 represents the *flingo* replacement of the rule in Lines 3 and 4 in Listing 11 which is responsible for generating possible integer attribute values.

```
1  &in{L..U} =: X :- include(X), type(X,T), integer(T),
   range(T,L,U).
2  &sum{X}    =  X :- include(X), type(X,T), integer(T), not range(T,_,_).
```

Listing 14. Generation of integer attribute values in *flingo*

The head atom `& inL..U =: X` assigns a values from the range `L..U` to variable `X`. Note that since variable `X` appears inside the theory atom, it is declared as an integer variables and therefore not subject to grounding. 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, `L` and `U`, are defined. In this case, this would not be necessary as both are constants; however, the current syntax of *flingo* only allows to use `&in` together with the assignment operator `=:`.

Due to the fact that integer variables in *flingo* are freed from grounding (and unlike in the *clingo* encoding), it is possible to reason with unbounded ranges (cf. Line 2). When no range is specified, the atom `&sumX = X` works similar to a choice rule in ASP in the sense that it defines variable `X` without constraining its range, thus effectively assigning any possible integer value to it.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

Next, Listing 15 shows some of the rules for Boolean constraint checking (compare with Listing 12).

```
1  defined(F) :- binary(F,X1,_,X2), &df{X1}, &df{X2}.

3  satisfied(F) :- binary(F,X1,"=", X2), &sum{X1} =  X2.
4  satisfied(F) :- binary(F,X1,">=",X2), &sum{X1} >= X2.
5  satisfied(F) :- binary(F,X1,"<=",X2), &sum{X1} <= X2.
```

Listing 15. Boolean constraint checking in *flingo*

In Line 1 we make use of *flingo*'s `&df` atom to check whether a variable is defined. Binary formulas are checked for satisfaction in the same way as before with the difference that we are using the *flingo* atom `&sum`. Recall that since integer variables are treated in a special way in *flingo*, they may only be accessed in the context of theory atoms (or constraints formed by them as in the case of variable `X2`).

Lastly, in Listing 16 we show rules for numerical calculations in *flingo* which correspond to the rules from Listing 13 for *clingo*.

```
1  &sum{ 1,X : set(P,X), include(X) } = F :- function(F,"count",P).
2  &sum{   X : set(P,X), include(X) } = F :- function(F,"sum",
   P).
3  &sum{ XL;  XR } = F :- binary(F,XL,"+",XR).
```

Listing 16. Numerical calculations in *flingo*

They represent calculations of `count` and `sum` aggregates, as well as for the + operator. While in the earlier encoding a different *clingo* functionality was used for each rule, here we use the *flingo* theory atom `&sum` for all calculations. For the `count` function, we utilize the `&sum` atom with weight 1 and for the + operator, we explicitly specify the two variables we want to add. For example, the atom `&sum 1,X : set(P,X), include(X) = F` assigns to variable `F` the number of variables `X` that are contained in set `P` and included in the solution.

As all these `&sum` atoms appear in the heads of rules, they assign values to variables. Note, however, that here we are not using the directional assignment operator `=:` but just normal equality. In general, *flingo* offers two distinct semantics: one, which is evoked by using the operator `=:`, only assigns a value to the variable on the right side if all variables on the left side are defined. The second one, evoked by using normal equality, has no directionality and forces all variables occurring in the constraint to take on a value. In that case, if both variables in the constraint `&sumX = Y` were undefined, the constraint would force them to take any value as long as they are equal (which could possibly result in an infinite amount of solutions).

In our case, the rules would behave the same under both semantics because we are already checking for definedness of the variables on the left side by means of the `include(X)` atom. This is possible due to the conditionality of the aggregate functions mentioned earlier.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

We can solve a Coom instance using *flingo* by adding the `-solver flingo` (or `-s flingo` for short) option on the command-line. By default, the CoomSuite uses *clingo* as solver. We do not show the command-line output here as it coincides with the one from *clingo*.

4.4 Adding user input

All the Coom language fragments described so far equip the user with the ability to build up a *configuration model* which can be seen as a blueprint for all possible configurations. Another important concept in configuration is that of *user requirements* which allows the user to specify knowledge that needs to be included in the current configuration (Soininen *et al.* 1998). For example, the user could require the solution to include an optional component (e.g., the basket of a bike) or to set the value for an attribute (e.g., defining a minimum number of storage space in the *TravelBike*).

The CoomSuite allows to perform this via a separate, so-called *user input* file that can be passed along on the command-line. For these means, the Coom language provides two keywords: `add` (Line 1) and `set` (Line 2–3) which respectively, add an object to the configuration and set the value of an attribute. For both, the variable identifiers from the refinement phase introduced in the previous section (cf. Section 4.2.1) have to be used, however, without the quotation marks. An example of such a user input for the *TravelBike* can be seen in Listing 17.

```
1  add frame[0].bag[0]
2  set color[0] = Yellow
3  set requestedVolume[0] = 200
```

Listing 17. A Coom user input file for the *TravelBike*

During parsing and the refinement phase, the Coom user input is converted into `user_include/1` and `user_value/2` predicates, respectively. Then, before solving, the CoomSuite checks if the user input is valid with respect to the configuration model and gives out warnings in case of inconsistencies. Warnings are given out when:

1. The referenced variable does not exist.
2. A value is being set for a variable not corresponding to an attribute.
3. The value is outside of the attribute domain.

Note that as upper bounds of cardinalities are compiled away during the refinement process, it is not necessary to check for exceeding maximal cardinalities. Instead, this is implicitly handled by the first kind of warning.

Going back to our simplified *TravelBike* from Listing 5, note that the product model does not have a `color` feature, and thus the CoomSuite would give out a warning of the first case about Line 2 in Listing 17.

Independently of any warnings, the user input is processed by the encoding in Listing 18. However, before adding any user input to the final configuration, the encoding first verifies for each `user_include/1` and `user_value/2` that the user input is consistent with the configuration model (Lines 1–3). We omit here the rules for integer domains as the check is similar as for discrete domains.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

```
1  consistent(X)   :- user_include(X), type(X,_).
2  consistent(X,V) :- user_value(X,V), type(X,T),
3                     discrete(T), domain(T,V).

5  include(X) :- user_include(X), consistent(X).
6  include(X) :- user_value(X,V), consistent(X,V).
7  value(X,V) :- user_value(X,V), consistent(X,V).
```

Listing 18. Solving user input in *clingo*

For each `user_include/1` predicate we check that the object to be added exists. For a `user_value/2` predicate we check that the attribute exists and that the value is in the domain of the attribute. Note that these checks only cover simple inconsistency cases and it is still possible for the user to provide inconsistent input such as an invalid combination of attribute values (which might be harder to detect).

Subsequently, in Lines 5–7 the consistent user input is added to the configuration. If the object in an `user_include/1` predicate refers to an attribute this assures that a value will be set for this attribute (but the user does not need to specify this value). Any user input not consistent with the configuration model is ignored.

We can solve a Coom instance with user input by using the `--user-input` option (or `-u` for short) and passing the path to the user input file as an argument. In Listing 19 we are running the simplified *TravelBike* example from before with the user input from Listing 17. The CoomSuite prints out a warning for the `set color[0] = Yellow` directive, as the variable is not part of the configuration model but proceeds to solve the configuration problem, ignoring the incosistent user input.

```
$ coomsuite solve travel-bike-simplified.coom -u user-input.coom -o coom
WARNING:  - Invalid user input.
Variable root.color[0] does not exist.
COOM Suite version 0.1
Reading from refined-input.lp
Solving...
Answer: 1
carrier[0]  frame[0]
requestedVolume[0] = 200    totalVolume[0]     = 200
carrier[0].bag[0] = "B50"   carrier[0].bag[0].volume[0] = 50
carrier[0].bag[1] = "B50"   carrier[0].bag[1].volume[0] = 50
carrier[0].bag[2] = "B50"   carrier[0].bag[2].volume[0] = 50
frame[0].bag[0]   = "B50"   frame[0].bag[0].volume[0]   = 50
SATISFIABLE
```

Listing 19. Solving the simplified *TravelBike* example together with user input

4.5 Reasoning with unbounded cardinalities

In all earlier Coom language fragments a lower and upper bound for feature cardinalities had to be explicitly specified (or the cardinality defaulted to 1..1 if no bounds were given). In practice, however, when modeling a configuration problem, it is often not possible to determine the exact (or even approximate) number of objects needed in

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

advance (Taupe *et al.* 2016). In this section we extend Coom[p] and Coom[x] to support unbounded cardinalities and call these Coom[p*] and Coom[x*], respectively. Then, we present a simple modification of our existing workflow to solve problems of such kind.

Listing 20 shows the *CargoBike*, a slightly modified version of the *TravelBike* example from Listing 5 such that the number of possible bags is now unknown.

```
1  product {
2                num   0-200    totalVolume
3                num   0-200    requestedVolume
4      0..*      Bag            bags
5  }
6  enumeration Bag {
7      attribute num volume

9      small    = ( 12 )
10     large    = ( 20 )
11 }
12 behavior {
13     require sum(bags.volume) = totalVolume

15     require totalVolume >= requestedVolume
16 }
```

Listing 20. Representation of the *CargoBike* example in Coom

As in the *TravelBike*, the *CargoBike* has two numeric features `totalVolume` and `requestedVolume` where the former is computed by summing up the volume of all bags and the latter specifies a minimum for the former (computed) value. However, different from before, the feature `bags` has cardinality `0..*`, which stands for zero or more bags and is unbounded above. This can not be solved natively with the encoding presented in Section 4.2.

As a simple workaround, we set a maximum bound during the refinement phase and check whether this leads to a solution. The CoomSuite currently offers two search strategies to determine this maximum bound: A linear and an exponential one (Francescutto *et al.* 2021). We call this approach *Incremental Bounds* and proceed in the following manner (cf. Figure 3):

1. Parse the Coom model into the serialized ASP fact format.
2. Set the initial maximum upper bound to n (typically $n = 1$).
3. Refine and solve the configuration problem.
4. If no solution is found, determine a new maximum bound according to the chosen strategy and repeat from step 3.

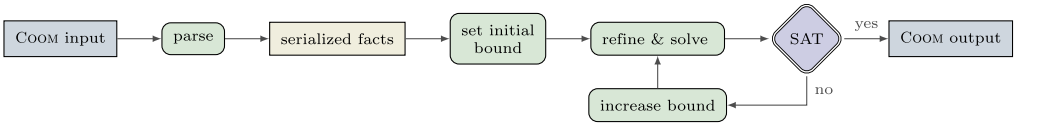


Fig 3. Workflow for incremental bounds algorithm.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

While the linear search strategy simply increases the maximum bound by a chosen value k (typically $k = 1$), the exponential one is inspired by Francescutto *et al.* (2021) and first doubles the current bound until finding a solution and then converges to the optimal bound.

While this approach is very simple, its advantages are that it is general (works with any combination of open cardinalities) and does not require any changes to the ASP encoding. Further, at least in an approximate way, it provides a built-in minimization on the number of parts. However, it does not (yet) make use of *clingo*'s multi-shot solving capabilities, and therefore grounding has to performed again on every iteration.

To solve a Coom instance with incremental bounds, we can use the `--bounds` option to specify the desired search strategy as shown in Listing 21. Here, we have chosen linear search and set the value of `requestedVolume` to 60 via a user input file. As the maximum volume of a single bag is 20, the iteration is not able to find a solution until the maximum bound is set to 3.

```
$ coomsuite cargo-bike.coom -u input.coom --bounds=linear -o coom

Solving with max_bound = 1
UNSATISFIABLE

Solving with max_bound = 2
UNSATISFIABLE

Solving with max_bound = 3
COOM Suite version 0.1
Reading from refined-input.lp
Solving...
Answer: 1
bags[0]
bags[1]
bags[2]
requestedVolume[0] = 60
totalVolume[0]     = 60

bags[0].size[0] = "large"
bags[0].size[0].volume[0] = 20  bags[0].size[0].weight[0] = 25

bags[1].size[0] = "large"
bags[1].size[0].volume[0] = 20  bags[1].size[0].weight[0] = 25

bags[2].size[0] = "large"
bags[2].size[0].volume[0] = 20  bags[2].size[0].weight[0] = 25

SATISFIABLE
```

Listing 21. Solving the *CargoBike* example with the incremental bounds option

5 Solving Coom interactively

While in the previous sections the workflow assumed an input from the user only at the beginning of the process, in practice, users typically want to interact with the system continuously, for example to experiment with different settings, explore the solution space or just to debug the configuration model. This is especially true for complex configuration problems where the number of solutions is large and difficult to envision by just looking

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

at an input model (whether in textual or graphical form). In that sense, a user interface (UI) for solving Coom interactively is a valuable addition to the CoomSuite workbench.

For that purpose, we next present a prototypical UI which is generated and driven by ASP, more precisely by the *clinguin* system (Beiser *et al.* 2024). This system uses a simple design with dedicated predicates to define a UI and the behavior of user-triggered events, thereby greatly facilitating the specification of continuous user interactions with an ASP system. For details about the syntax and functionality of *clinguin* we refer the reader to (Beiser *et al.* 2024).

The UI presented in this section uses the Coom[x] encoding presented in Section 4.2 plus an additional *clinguin* encoding defining its layout, style and functionality. Since integration with constraint systems such as *flingo* is currently limited in *clinguin*, the UI only works with *clingo*.

User input as introduced in Section 4.4 is no longer needed. Instead, the user can interactively make choices to create a configuration solution. For this, we use assumptions, which can be interpreted as integrity constraints that force the encoding to entail the provided atom. As before with the user input, these assumptions can be used to set values to attributes or to force the inclusion of a part in the final configuration. Moreover, the UI provides additional functionality for the user to browse solutions, download the current Coom solution and obtain basic explanations of why a selection is not valid.

We start by showcasing a simple interaction of the UI using the *TravelBike* example by presenting snippets of the UI encoding together with screenshots of the possible user interactions. The full UI encoding can be found in (Baumeister *et al.* 2025a). Then, we proceed to show the explanation features of the UI and how they are implemented in *clinguin*.

Furthermore, to better demonstrate the UI capabilities, we do not use the simplified version of the *TravelBike* from above but instead use the complete example. The main differences are that the `Wheel` and `Bag` enumerations additionally have a `weight` attribute contributing to the `totalWeight` of the bike and which can be constrained by a user-set value `maxWeight`. There are also additional `feature`[s] such as the `color` of the bike and the `material` of the `Bag`. Lastly, the *TravelBike* contains some additional constraints among which we only highlight the conditional requirement in Listing 22 stating that the `color Red` implies a `frontWheel` of size `20`.

```
1  condition color = Red
2  require frontWheel.size = 20
```

Listing 22. Conditional requirement of the complete *TravelBike* example

Attributes and values Figure 4 displays the initial state of the UI upon loading with an opened dropdown menu showing the available options for the `color` of the bike. Upon selecting `Red` as the `color`, the UI updates to show any inferred values resulting from this choice. In this case, the constraint from Listing 22 causes a `frontWheel` of size `20` to be inferred and this is updated accordingly in the UI (cf. Figure 5).

We now explain how this is represented in the *clinguin* encoding. For the sake of brevity, we only show exemplary snippets of the encoding relevant to attributes and values. Note

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

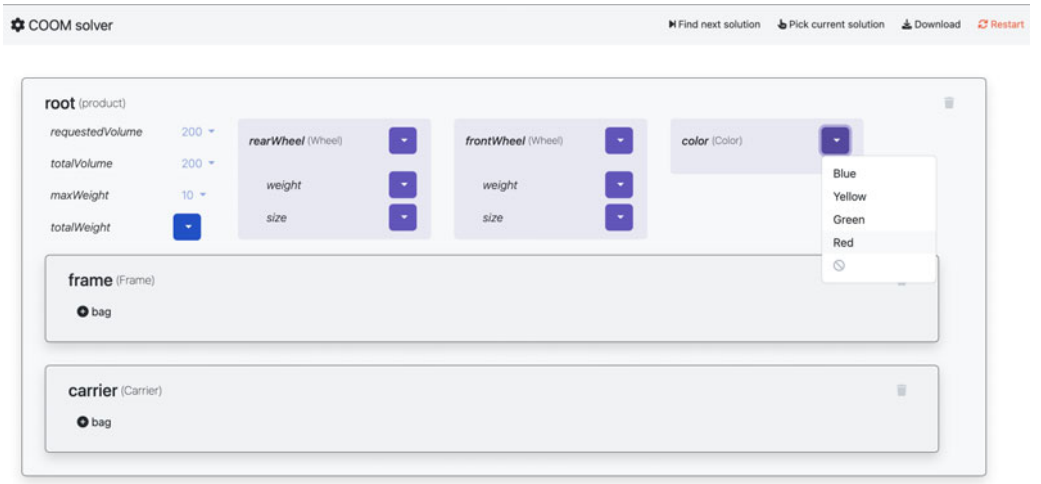


Fig 4. Initial state of UI.

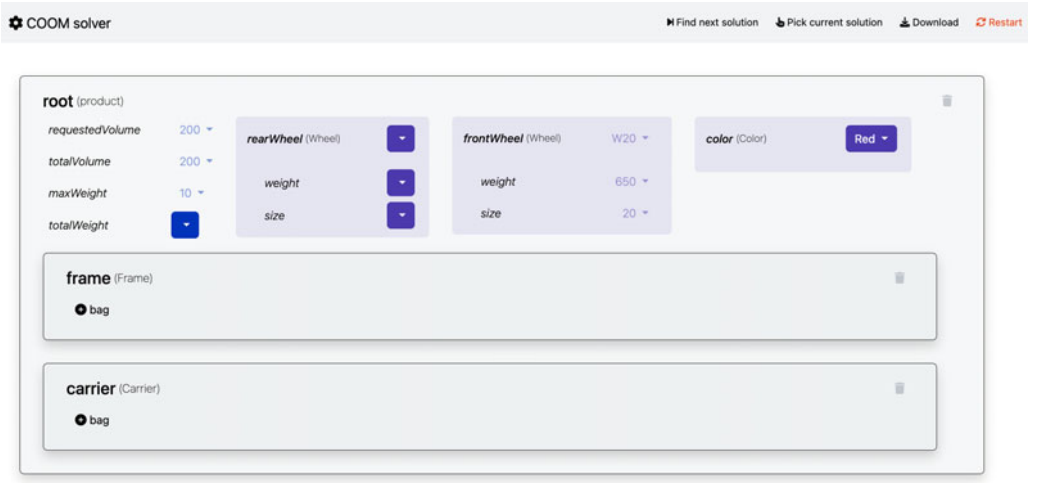


Fig 5. UI after selecting the `color red`.

that while the UI is built to represent the Coom input language, the encoding is using the predicates (and naming conventions) of the ASP fact format (cf. Section 4.2).

The first two Lines of Listing 23 create auxiliary predicates for the two types of attributes in our fact format. Subsequently, these are used in Lines 3 and 4 to create

```
1  attr_type(X,discrete) :- type(X,T), discrete(T).
2  attr_type(X,integer)  :- type(X,T), integer(T).
3  i_attr(X,AT) :- attr_type(X,AT), _all(include(X)).
4  i_attr(X,AT) :- attr_type(X,AT),      include(X), _clinguin_browsing.
5  i_value(X,V) :- i_attr(X,_), _all(value(X,V)).
6  i_value(X,V) :- i_attr(X,_),      value(X,V), _clinguin_browsing.
```

Listing 23. Definition of included attributes and their values in the UI

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

the predicate `i_attr(X,AT)` stating that attribute variable X of type AT is included (in the solution).

Here, Line 3 checks whether an attribute variable is included in all solutions using the dedicated *clinguin* predicate `_all/1`. This allows us to show only relevant attributes in the UI. In the two screenshots, for instance, we do not see any attributes for the possible `bag`[s] as they are not included in all solutions (thus not mandatory). Line 4, on the other hand, checks what attribute variables are included while browsing through solutions. Similarly, Lines 5 and 6 define the (attribute) values to be shown via predicate `i_value(X,V)`.

With these auxiliary predicates, the first rule in Listing 24 creates a dropdown menu on the UI for each attribute, and the second rule adds the text for this dropdown menu when a value exists. We can see this when comparing the first and second screenshot in Figures 4 and 5 where the text in the dropdown menus of the `color` and `frontWheel` enumerations changes accordingly.

```
1  elem(dd(X), dropdown_menu,
2               attr_container(X)) :- i_attr(X,_).
3  attr(dd(X),selected,V)          :- i_value(X,V).
```

Listing 24. Definition of UI dropdown menus

The style of the dropdown menus is defined in Listing 25 where the first two rules set the style for inferred values. The first rules add the light text with low opacity for values that are inferred but not (yet) selected, for example as for the `frontWheel`. Furthermore, the second rule removes the border of the dropdown menus of such attributes to better distinguish them.

Next, the third and fourth rule add the class `"btn-secondary"`, which is the purple button style, for mandatory discrete attributes (stemming from enumerations in Coom) for which either a value has been selected or there are multiple options left. Note that for numeric values (not shown here) this works similarly but with the class `"btn-primary"`, which is the blue button style. Selecting values in *clinguin* is done via adding assumptions to *clingo* for the corresponding atom and the predicate `_clinguin_assume/2` is added by *clinguin* for every atom that has been assumed to be true or false.

```
1  attr(dd(X),class,("fw-light";
2                    "opacity-50"))  :- i_value(X,V), _all(value(X,V)),
3                                       not _clinguin_assume(value(X,V),true).
4  attr(dd(X),class,"border-0")       :- i_attr(X,discrete), i_value(X,_),
5                                       not _clinguin_assume(value(X,_),true).
6  attr(dd(X),class,"btn-secondary") :- i_attr(X,discrete),
7                                       not i_value(X,_).
8  attr(dd(X),class,"btn-secondary") :- i_attr(X,discrete),
9                                       clinguin_assume(value(X,_),true).
```

Listing 25. Definition of UI style for attributes

The rules in Listing 26 are responsible for providing the options of the dropdown menus that are still valid. The first rule creates an auxiliary predicate with all possible values

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

which is subsequently used in the last three rules to create one dropdown menu item for each such value.

```
1  v_option((X,V),valid) :- i_attr(X,_), _any(value(X,V)),
2                           not _clinguin_assume(value(X,_),true).
3  elem(ddi(X,V), dropdown_menu_item, dd(X))  :- v_option((X,V),_).
4  attr(ddi(X,V), label, V)                   :- v_option((X,V),_).
5  when(ddi(X,V), click, call,
6                 add_assumption(value(X,V))) :- v_option((X,V),_).
```

Listing 26. Definition of possible values for UI dropdown menus

Upon clicking on one of these values, a call is made to the solver to add the assumption of the corresponding value assignment, thus forcing the entailment of the given value and restricting the possible solutions (cf. Line 6). The dedicated *clinguin* predicate `_any/1` is the counterpart for `_all/1` and specifies that an atom is contained in at least one stable model.

Explanations The two cropped screenshots in Figure 6 show the explanation features of the UI. Recall the constraint in Listing 22 stating that the `color Red` implies a `frontWheel` of size `20` (which belongs to option `W20`) and that the UI inferred the latter value upon selecting `Red` as the `color`. Accordingly, when we open the dropdown menu of the `frontWheel`, we see the other (invalid) options in red but it is still possible to click on them. When selecting an invalid value, however, the affected values of the `frontWheel` and the `color` turn red and the UI shows an explanation of why the newly selected value is not valid. We now explain how this works inside *clinguin*.

Our approach to explanations for invalid values follows (Beiser *et al.* 2024). In a nutshell, this means that we use a specialized *clinguin* backend which provides a minimal unsatisfiable set (MUS) of assumptions whenever the encoding is unsatisfiable. In this way, we allow the user to select invalid values, and thus to let the problem become unsatisfiable, while subsequently using the information about faulty assumptions to highlight previously selected values in red. For our approach, this has the limitation that it only works for invalid selections of attribute values but not for the inclusion of an invalid part.

Additionally, we leverage the definition of constraints in the refined ASP instance to provide the user with a natural language explanation of why a certain value is invalid. To do this, we indicate to the explanation backend that the atoms of predicate `constraint/2` should be treated as assumptions, and thus be included as part of the reasons for unsatifiablity. Moreover, we extend the ASP fact format for the configuration to include the predicate `explanation/2` to store natural language explanations for Boolean constraints. For instance, the constraint `!root.color[0] = Red||root.frontWheel[0].size[0] = 20` has a natural language explanation: *"If the color is red, then the size of the front wheel should be 20."*

Thanks to the generality of Boolean operators, we were able to automatically generate these explanations for all Boolean constraints using a Large Language Model (LLM)

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

by prompting it with two simple examples. However, they can also be provided by the user. In fact, Coom offers a dedicated keyword for this which we plan to integrate in the workflow of the CoomSuite in the future. While it is desirable to have these kinds of explanations for all constraints, we limit ourselves to Boolean constraints for now and leave explanations for table constraints as future work as their treatment is more complex.

A snippet of the corresponding UI encoding is shown in Listing 27.

```
1  attr(ddi(X,V),class,
2                ("text-danger")) :- i_attr(X,_), type(X,T),
3                                    domain(T,V), not _any(value(X,V)).

5  attr(dd(X), class, ("btn-danger")) :- _clinguin_mus(value(X,V)).
6  elem(m(C), message, window) :- _clinguin_mus(constraint(C,"boolean")).
7  attr(m(C), message, M)      :- _clinguin_mus(constraint(C,"boolean")),
8                                 explanation(C,M).
```

Listing 27. UI explanation encoding

The first rule sets the red text style for the invalid dropdown menu items which belong to domain values not contained in any solution. The other three rules are utilizing the dedicated *clinguin* predicate `_clinguin_mus/1` which provides information about the MUS. While the first rule sets the red button style for all values contained in the MUS, the second and third rule provide the (natural language) explanation inside a separate window (cf. Figure 6).

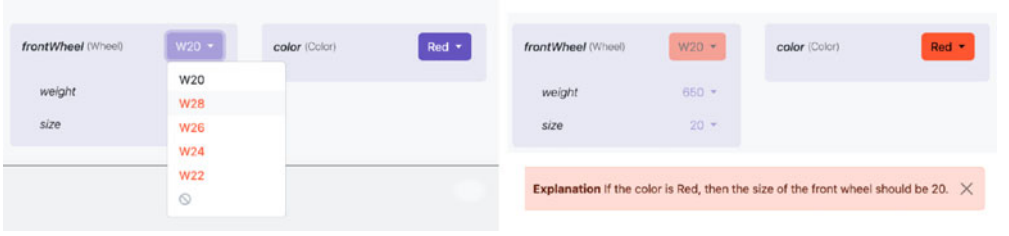


Fig 6. Explanations in the UI after selecting an invalid value.

Finally, leaving behind the topic of explanations, the last screenshot shows the addition of a `bag` to the configuration (cf. Figure 7) by means of clicking on the small icon containing a +. Since it was optional to add this part, it includes a red button to remove it. For the sake of brevity, we do not go into further detail here. As next steps, the user might decide to browse the different solutions, pick one and continue modifying it, or download it in the format of a Coom solution. The interested reader can try this out by following the instructions in (Baumeister *et al.* 2025a). Currently, this requires running *clinguin* separately but for future versions of the CoomSuite a direct integration is planned.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

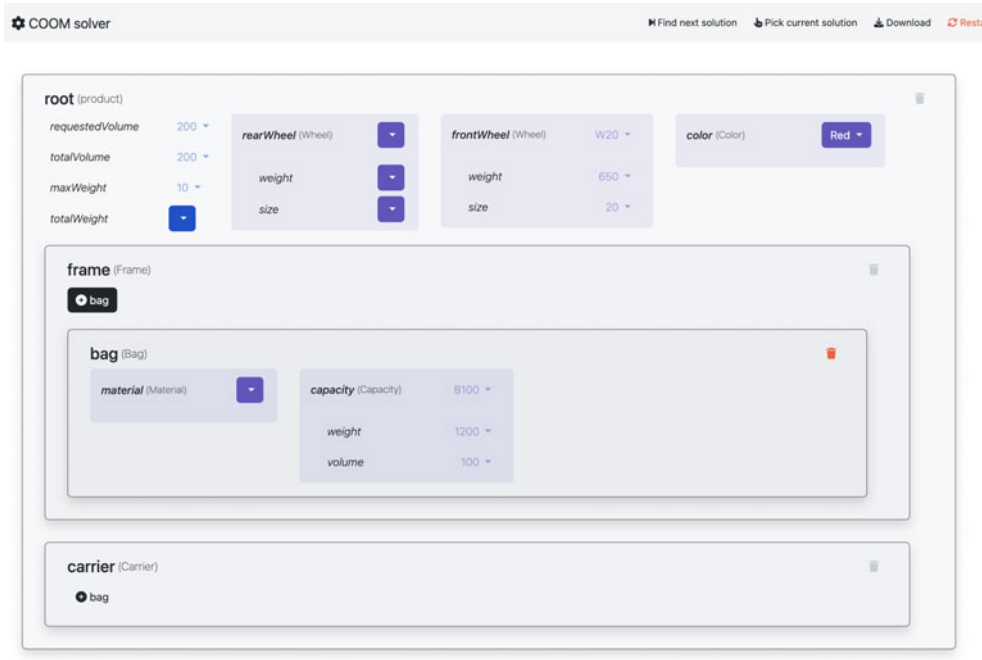


Fig 7. Adding a bag to the solution.

6 The CoomSuite Workbench

The CoomSuite (Baumeister *et al.* 2025a) is intended to serve as a workbench for experimentation with industrial-scale product configuration problems. While the included benchmark collection can be utilized with other paradigms, its current infrastructure is primarily geared towards ASP. Specifically, the CoomSuite is available as a Python package, installable via pip. It includes a (customizable) ANTLR v4 parser to convert Coom specifications into facts, along with an ASP encoding to harmonize the fact format with the chosen configuration encoding. The current distribution includes four scalable benchmark series, a single ASP encoding covering all essential configuration concepts needed for the three Coom language fragments, as well as an additional one for the hybrid solver *flingo* (Cabalar *et al.* 2025).

For each of the three Coom language fragments, we include a benchmark set in the CoomSuite detailed in Table 1. The two language extensions Coom[p*] and Coom[x*] currently have no dedicated benchmark sets. Additionally, the CoomSuite includes the benchmark set of a *Restaurant* corresponding to the Coom[x] language, however, with

Table 1. *Benchmark sets of the CoomSuite*

Benchmark set	Language	Description	Scalable factor	# Instances
Core	CoomCore	Random table constraints	#Attributes, #Values	45
CityBikeFleet	Coom[p]	Fleet of *CityBikes*	#Bikes	15
TravelBikeFleet	Coom[x]	Fleet of *TravelBikes*	#Bikes	15

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

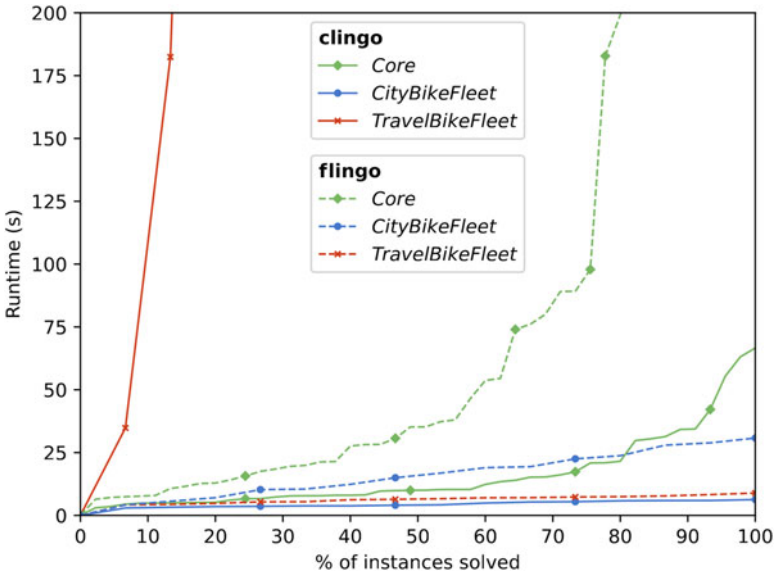


Fig 8. Runtimes for benchmarks of Table 1.

much simpler numerical calculations. Here, the aim is to configure the assignment of a given number of chairs to tables of different sizes. It makes use of partonomy as well as simple numeric constraints and the scalable factor is the total number of chairs needed. We decided to not show the results of the *Restaurant* domain here, as they were similar to the *Core* and *CityBikeFleet* domains.

To provide a glimpse into the usage of the CoomSuite, we conducted sample benchmarks comparing the standard ASP encoding (Section 4.2.2) with that designed for *flingo* (Section 4.3), which utilizes integer variables. We ran all instances of the benchmarks in Table 1 on a compute cluster with Intel Xeon E5-2650v4@2.9 GHz CPUs with 64 GB of memory running Debian Linux 10.[5] We used a timeout of 300 s and limited the memory to 16 GB per instance. Figure 8 shows the runtimes of finding one stable model for both solvers *clingo* and *flingo*. On the two non-numeric domains (*Core* and *CityBikeFleet*) *clingo* performs better than *flingo*. While *clingo* is able to solve most instances within a couple seconds (and takes up to 60 s for bigger instances), *flingo* times out on some of the instances of the *Core* domain. In general the runtimes of *flingo* are about one order of magnitude higher than those of *clingo*. However, for the *TravelBikeFleet*, *flingo* clearly outperforms *clingo* due to the problem of the latter with large numeric ranges. In contrast, *flingo* can handle these effortlessly due to its native handling of integer variables. Note that when *clingo* times out it is usually during grounding, while for *flingo* this usually happens during its internal preprocessing which uses *clingo*'s Python API.

7 Discussion

Researchers often lack access to industrial-scale examples to evaluate their work due to confidentiality or limited public availability. To address this challenge in product

[5] https://www.cs.uni-potsdam.de/bs/research/labs.html#hardware

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

configuration, we introduce the CoomSuite, a workbench offering a curated collection of product model benchmarks and tools for converting them to ASP. These benchmarks, derived from industrial contexts, reflect the key challenges of product configuration. The workflow includes a refinement step that separates the ASP representation of the Coom input from the representation tailored to specific ASP encodings or systems. We highlighted the design of a series of such ASP encodings that handle increasingly complex Coom models.

Our work provides not only the first publicly available ASP implementation of Coom but also (indirectly) establishes first semantic underpinnings for Coom. However, this is just the starting point. Future challenges include addressing unimplemented Coom features such as optimization and explanations, as well applying and further developing alternative ASP configuration encodings (Falkner *et al.* 2015; Gençay *et al.* 2019; Rühling *et al.* 2023) within this uniform industrial-scale setting. Concrete improvements include a normalization of constraints during parsing which enables the reutilization of information during solving, studying a more native representation of Coom for *flingo*, and the development of a truly incremental approach for unbounded cardinalities using *clingo*'s multi-shot solving capabilities. While the current approach to the latter provides a simple and general baseline solution, a multi-shot approach will most likely provide a much better performance. However, this requires greater modifications to the ASP encoding and further study on how to handle multiple, especially nested, open cardinalities effectively as well as an extension of the benchmark sets for the sake of an exhaustive evaluation.

As what concerns transparency and user integration, we can leverage ASP-driven visualization (Hahn *et al.* 2022) and enhance the capabilities for interactive exploration of the configuration space by means of further conceptual and technical development of the UI.

We also aim to expand the model benchmarks in the CoomSuite with more domains inspired by real-world applications, creating a comprehensive and challenging workbench for advancing ASP development and beyond.

References

Aerts, B., Deryck, M. and Vennekens, J. 2022. Knowledge-based decision support for machine component design: A case study. *Expert Systems with Applications* 187, 115869. https://doi.org/10.1016/J.ESWA.2021.115869

Antkiewicz, M., Bak, K., Murashkin, A., Olaechea, R., Liang, J. and Czarnecki, K. 2013a. Clafer tools for product line engineering. *In 17th International Software Product Line Conference Co-Located Workshops, SPLC 2013 Workshops, Tokyo, Japan – August 26 – 30, 2013*, ACM, 130–135. https://doi.org/10.1145/2499777.2499779

Antkiewicz, M., Bak, K., Murashkin, A., Olaechea, R., Liang, J. and Czarnecki, K. 2013b. Clafer configurator. URL: https://github.com/gsdlab/ClaferConfigurator

Banbara, M., Kaufmann, B., Ostrowski, M. and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming* 17, 4, 408–461. https://doi.org/10.1017/S1471068417000138

Baumeister, J. 2018. Evaluation of configuration knowledge in industrial manufacturing. In *Proceedings of the Conference "Lernen, Wissen, Daten, Analysen", LWDA'18*, R. Gemulla, S. Ponzetto, C. Bizer, M. Keuper, and H. Stuckenschmidt, Eds. Vol. 2191 of *CEUR Workshop Proceedings,* CEUR-WS.org, 38–48. https://ceur-ws.org/Vol-2191/paper5.pdf

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

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

Baumeister, J., Herud, K., Reutelshöfer, J. and Belli, V. 2025b. Coom language. URL: https://www.coom-lang.org/

Beiser, A., Hahn, S. and Schaub, T. 2024. Asp-driven user–interaction with clinguin. In *Technical Communications of the Fortieth International Conference on LogicProgramming (ICLP'24)*, P. Cabalar and T. Swift, Eds, EPTCS.

Brunner, J.-S., Ma, L., Wang, C., Zhang, L., Wolfson, D. C., Pan, Y. and Srinivas, K. 2007. Explorations in the use of semantic web technologies for product information management. In *Proceedings of the 16th International Conference on World Wide Web, WWW*, ACM, pp. 747–756. https://doi.org/10.1145/1242572.1242673

Cabalar, P., Kaminski, R., Ostrowski, M. and Schaub, T. 2016. An ASP semantics for default reasoning with constraints. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, S. Kambhampati, Eds. IJCAI/AAAI Press, 1015–1021. https://doi.org/10.5555/3060621.3060762

Cabalar, P., Fandinno, J., Schaub, T. and Wanko, P. 2020. An ASP semantics for constraints involving conditional aggregates. In *Proceedings of the Twenty-Fourth European Conference on Artificial Intelligence (ECAI'20)*, IOS Press, 664–671. https://doi.org/10.3233/FAIA200152

Cabalar, P., Fandinno, J., Schaub, T. and Wanko, P. 2025. flingo. URL: https://github.com/potassco/flingo

Cabalar, P., Fandinno, J., Schaub, T. and Wanko, P. 2026. Flingo — Instilling ASP expressiveness into numerical constraints. *CoRR,* abs/2602.09620. URL: https://arxiv.org/abs/2602.09620

Carbonnelle, P., Vandevelde, S., Vennekens, J. and Denecker, M. 2023. Interactive configurator with FO(.) and IDP-Z3. *CoRR*, abs/2202.00343. URL: https://arxiv.org/abs/2202.00343

Coskun, G., Heese, R., Luczak-Rösch, M., Oldakowski, R., Paschke, A., Schäfermeier, R. and Streibel, O. 2009. Towards a corporate semantic web. In *Proceedings of the 5th International Conference on Semantic Systems*, A. Paschke, H. Weigand, W. Behrendt, K. Tochtermann, and T. Pellegrini, Eds. Verlag der Technischen Universität Graz, 602–610. URL: http://www.i-semantics.at/2009/papers/towards_a_corporate_semantic_web.pdf

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

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

Falkner, A., Haselböck, A., Krames, G., Schenner, G., Schreiner, H. and Taupe, R. 2020. Solver requirements for interactive configuration. *Journal of Universal Computer Science* 26, 3, 343–373. https://doi.org/10.3897/jucs.2020.019

Felfernig, A., Hotz, L., Bagley, C. and Tiihonen, J. 2014. *Knowledge-Based Configuration: From Research to Business Cases*. Elsevier/Morgan Kaufmann. https://doi.org/10.1016/C2011-0-69705-4

Felfernig, A., Falkner, A., Atas, M., Erdeniz, S., Uran, C. and Azzoni, P. 2017. ASP-based knowledge representations for IoT configuration scenarios. In *Proceedings of the Nineteenth International Configuration Workshop (CONF'17)*, L. Zhang and A. Haag, Eds., IÉSEG éditions, 62–67.

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

Felfernig, A., Falkner, A. and Benavides, D. 2024. *Feature Models - AI-Driven Design, Analysis and Applications. Springer Briefs in Computer Science*. Springer-Verlag. https://doi.org/10.1007/978-3-031-61874-1

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

Gebser, M., Kaminski, R. and Schaub, T. 2011. aspcud: A Linux package configuration tool based on answer set programming. In *Proceedings of the Second International Workshop on Logics for Component Configuration (LoCoCo'11),* Electronic Proceedings in Theoretical Computer Science (EPTCS), Vol. 65 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 12–25. URL: https://doi.org/10.4204/eptcs.65.2

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

Hahn, S., Sabuncu, O., Schaub, T. and Stolzmann, T. 2022. Clingraph: ASP-based visualization. In *Proceedings of the Sixteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22)*, Vol. 13416 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 401–414. URL: https://doi.org/10.1007/978-3-031-15707-3_31

Herud, K., Baumeister, J., Sabuncu, O. and Schaub, T. 2022. Conflict handling in product configuration using answer set programming. In *Proceedings of the Fifteenth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'22)*, Vol. 3193 of *CEUR Workshop Proceedings. CEUR-WS.org.* URL: https://ceur-ws.org/Vol-3193/paper2ASPOCP.pdf

Hotz, L., Felfernig, A., Stumptner, M., Ryabokon, A., Bagley, C. and Wolter, K. 2014. *Configuration Knowledge Representation and Reasoning.* In Felfernig A. et al., Eds. chapter 6, 41–72. https://doi.org/10.1016/b978-0-12-415817-7.00006-2

Junker, U. 2006. Configuration. In *Handbook of Constraint Programming*, chapter 24, Rossi, F., van Beek, P. and Walsh, T., Eds. Elsevier Science, 837–873. https://doi.org/10.1016/s1574-6526(06)80028-3

Juodisius, P., Sarkar, A., Mukkamala, R.-R., Antkiewicz, M., Czarnecki, K. and Wasowski, A. 2019. Clafer: Lightweight modeling of structure, behaviour, and variability. *The Art, Science, and Engineering of Programming* 3, 1, 2. https://doi.org/10.22152/programming-journal.org/2019/3/2

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

Lin, J., Fox, M. S. and Bilgic, T. 1997. A product ontology. Enterprise Integration. URL: http://eil.utoronto.ca/wp-content/uploads/enterprise-modelling/papers/lin-eil97.pdf

Rühling, N., Schaub, T. and Stolzmann, T. 2023. Towards a formalization of configuration problems for ASP-based reasoning: Preliminary report. In *Proceedings of the Twenty-Fifth International Configuration Workshop (CONF'23)*, J. Horcas, J. Galindo, R. Comploi-Taupe, and L. Fuentes, eds. Vol. 3509, 85–94. *CEUR Workshop Proceeding.* URL: https://ceur-ws.org/Vol-3509/paper12.pdf

Soininen, T., Tiihonen, J., Männistö, T. and Sulonen, R. 1998. Towards a general ontology of configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 12, 4, 357–372. https://doi.org/10.1017/s0890060498124083

Taupe, R., Falkner, A. and Schenner, G. 2016. Deriving tighter component cardinality bounds for product configuration. In *Proceedings of the Eighteenth International Configuration Workshop (CONF'16)*, Vareilles, L. Hvam, C. Forza, and C. Becker,

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press

Eds. 47–54. URL: http://cp2016.a4cp.org/program/workshops/CWS-2016-Proceedings.pdf#section∗.8

Vandevelde, S., Vennekens, J., Jordens, J., Van Doninck, B. and Witters, M. 2024. Knowledge-based support for adhesive selection: Will it stick? *Theory and Practice of Logic Programming* 24, 3, 560–580. https://doi.org/10.1017/S1471068424000024

https://doi.org/10.1017/S1471068426100404 Published online by Cambridge University Press