A Framework for Categorizing Product Configuration Systems (Position Paper)

Joachim Baumeister[1,2,*], Konstantin Herud[2], Lukas Ley[2] and Jochen Reutelshöfer[2]

[1]University of Würzburg, Am Hubland, Würzburg, Germany
[2]denkbares GmbH, John-Skilton-Straße 8, Würzburg, Germany

Abstract

Despite significant progress in smart product configuration systems, driven by automation and technological advancements, a key challenge persists: the absence of a standardized framework for evaluating those systems with respect to the richness of the used knowledge representation language. We tackle this challenge by introducing a framework of *profiles*, that categorizes and compares different systems' expressiveness and practical applicability in industrial settings.

Keywords

Smart Product Configuration, Knowledge Engineering, Knowledge Representation

1. Introduction

The challenge of *configuration* has been a fundamental aspect of expert systems (XPS) [1, 2] since the 1980s. In recent years, *product configuration systems* have seen a significant surge in interest, driven by advancements in automation and the rise of Industry 4.0 within mechanical engineering and production domains [3, 4, 5]. A variety of tools and methodologies have emerged in the market to efficiently support the product configuration process, often integrated in so-called Configure-Price-Quote (CPQ) tools. However, a thorough market analysis reveals a significant challenge: It is difficult to compare these tools due to the lack of a comprehensive benchmark for assessing performance. The combinatorial nature of configuration problems makes the solving task a challenge at scale. Moreover, the effectiveness of these tools is not solely dependent on performance; the expressive power of their knowledge representation language also plays a crucial role.

Reflecting on the past, the development and innovation of early Expert Systems (XPS) achieved significant advancements by distinguishing the problem-solving method from the knowledge base and its acquisition, as illustrated in Figure 1. A problem-solving method in Expert Systems describes an inference engine to solve a given problem context.

The purpose of this position paper is to revive this approach by outlining a conceptual framework that identifies the essential tasks involved in product configuration knowledge and organizes them into a series of definable properties. By introducing these so-called *profiles*, we aim to make engines and methods comparable once more, both in terms of their expressiveness and their practical applicability in real-world scenarios.

LWDA'24: Lernen, Wissen, Daten, Analysen. September 23–25, 2024, Würzburg, Germany
*Corresponding author.
joba@uni-wuerzburg.de (J. Baumeister)

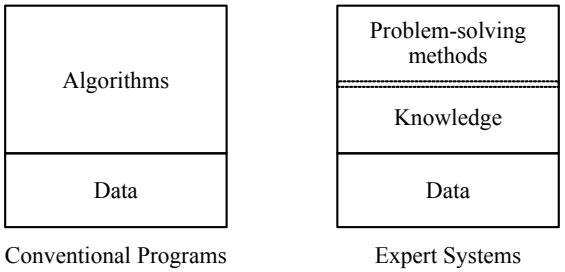

Figure 1: Separation of problem-solving methods and knowledge base, Puppe [2].

This framework is grounded in extensive experience from consulting projects that focus on implementing smart product configuration systems within the industrial engineering and production domain. For the configuration task, we propose *smart product configuration* as an advanced problem-solving task, extending traditional frameworks to meet contemporary needs.

A similar research has been started by Tiihonen [6], where PCML as a declarative language for product configuration knowledge with a formal semantics was introduced and applied to industrial knowledge bases. Here, the complexities of the particular knowledge bases are listed, e.g., percentage of cardinality use, the size of the value domains, and total number of features. However, no categorization of the knowledge elements is discussed, neither are the requirements of smart configuration considered. The introduction of OWL2 profiles [7, 8] followed a similar idea: Here, different OWL2 profiles are defined for fractions of the semantic web language OWL in order to support the knowledge engineering process and the reasoning support. Each OWL2 profile covers a specific collection of expressions.

2. Smart Product Configuration

In comparison to classic product configuration systems, a smart product configuration system always remains in the space of valid solutions. We first introduce the solution space for product configuration systems.

2.1. The Solution Space in Product Configuration

In product configuration, the term *solution space* denotes the set of all possible configurations that satisfy the requirements and constraints given in the knowledge base. For n features in a knowledge base, the solution space can be thought of as an n-dimensional space. Each dimension of this space belongs to the value domain of a feature. We illustrate this by an example with three features in Figure 2.

In Figure 2 we illustrate the solution space with a small example based on the configuration of a bike. Here, for the sake of simplicity only the three features *Color*, wheel *Size*, and *Bag* design can be configured by the user. The left side of Figure 2 shows the solution space of possibly producible bikes without any constraints, i.e., $5 \times 5 \times 4 = 100$ possible configurations are depicted. The right side shows the solution space considering a number of constraints, restricting the number of valid configurations. Configurations that satisfy all constraints are

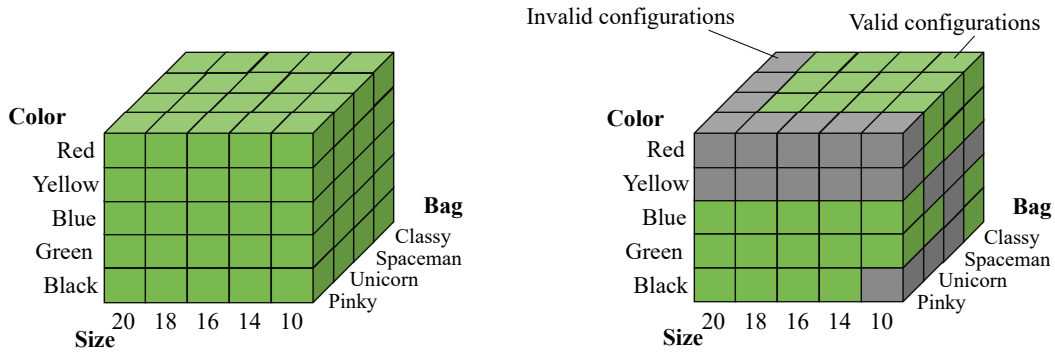

Figure 2: The whole solution space for three configurable features (left) and the restricted solutions space based on constraints (right).

shown in green. Configurations that violate any of the constraints and are thus invalid, are shown in gray. Consider as an example for a constraint the right front corner of the right cube. We see that pinky bags and wheel size 10 is only allowed for blue and green colored bikes.

In the following, we motivate that *smart configuration systems* need to be aware of the solution space at any time of the configuration process. Within the solution space, systems need to focus on optimizing certain aspects such as cost, performance, or manufacturing efficiency. Recent smart configuration systems navigate the solution space efficiently to find the optimal configurations given these criteria.

2.2. Elements of Smart Product Configuration

In comparison to a classic configuration system, a *smart product configuration* system offers a number of benefits to the user:

Flexible and Secure Configuration: Users have the freedom to customize their product by selecting features in any order they prefer, while ensuring all configurations remain valid. Once a user selects a feature, the system will not change it unless explicitly approved by the user. The status of the configuration depends solely on the number of features selected, not the order in which they are chosen. This is possible due to the declarative nature of smart configuration knowledge bases in contrast to procedural statements in classical configuration systems.

Interactive Configuration Process: During the configuration process, the system displays available options for each feature and identifies which choices are currently blocked, because they would violate constraints. This ensures the user remains within the solution space, i.e., the space of valid product configurations.

Conflict Resolution and Explanations: If a desired value is blocked due to a potential conflict with existing choices, then the system suggests modifications to previous selections that could resolve the conflict. This allows for the inclusion of the desired feature while

maintaining a valid product configuration. Additionally, the system provides detailed explanations for why certain configurations are not permissible, helping users understand the constraints.

Intelligent Default Settings: The system establishes default settings for features that the user has not yet configured, boosting the initial setup process and enhancing the user experience.

In summary, a smart configuration system places much greater demands on the engine than a classic configuration system. For example, a procedural evaluation of the knowledge and the consideration of the evaluation sequence is no longer possible. In addition, a smart configuration system must be able to offer a valid configuration at all times, including at the beginning of a configuration session.

3. Product Configuration Profiles

Product configuration systems are built by using a knowledge representation language. We introduce the open and textual language COOM for defining product configuration knowledge bases. COOM stands for "COnfiguration Object Model" and is described as an open standard language [9]. This language is intended to serve as an interface between knowledge bases from different origins and reasoning engines with varying specialization.

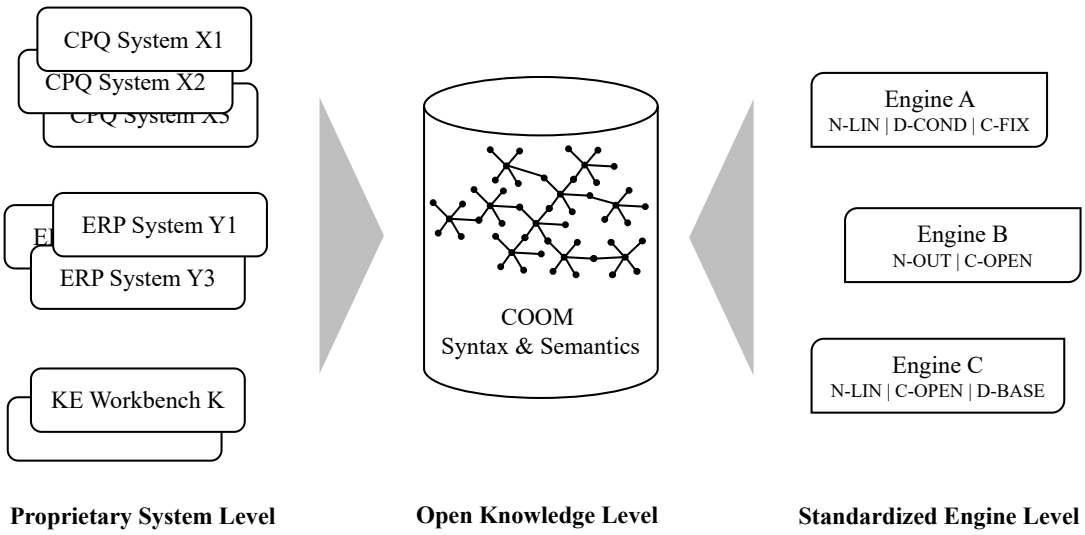

Figure 3: Integration of the COOM language into the product configuration workspace.

A typical scenario for using the COOM language in an application scenario is illustrated in Figure 3: The knowledge developed in proprietary product configuration systems is compiled into the open COOM syntax. In COOM the required profiles are identified and a suitable

reasoner can be selected. A corresponding reasoner then is linked to the original CPQ system or another end-user interface. Besides commercial implementations of COOM profiles, there already exists an open-source engine of COOM [10]. We sketch the elementary blocks of the COOM language in the following.

3.1. Basic Syntax of COOM

Here, we introduce the most important concepts of the language by example. For a complete and thorough introduction to the language we refer to `coom-lang.org`.

In the following we illustrate essential parts of the COOM language with examples from the bicycle domain. We define a "PaperKidsBike"[1] where customers are only allowed to configure the color of the bike, the size of the wheels, the version of the two bags, and whether the bike should have support wheels or not. Also, customers are able to enter the weight of the bike's driver.

The knowledge base is organized as a hierarchical structure of feature elements, which together form a comprehensive representation of the product to be configured. In the example, we see that the markup `product` defines an instance `kidsBike` of the structure type `PaperKidsBike`. Recursively, the `PaperKidsBike` type is defined by the enumeration types `Color`, `Bag` and `Wheel` and a boolean type representing the option support wheels, plus a numerical type holding the weight of the driver. The type `Bag` has fixed cardinality 2, i.e., exactly two bag instances are attached to the bike.

```
1  product {
2      PaperKidsBike kidsBike
3  }
4
5  structure PaperKidsBike {
6      Color  color
7    2 Bag    bag
8      Wheel  wheelSize
9      bool   supportWheel
10     num    driversWeight
11 }
12
13 enumeration Wheel { W14 W16 W18 W20 }
14
15 enumeration Color { Red Yellow }
16
17 enumeration Bag   { Classy Spaceman Unicorn Pinky }
18
19 behavior PaperKidsBike {
20     condition color = Yellow
21     require  wheelSize = W18 || wheelSize = W20
22 }
23
24 behavior PaperKidsBike {
```

[1]The PaperKidsBike is an extended version of the KidsBike model (https://www.coom-lang.org/profile_core).

```
    combinations (  supportWheel   wheelSize   driversWeight )
    allow        (  true           (W14,W16)   <30           )
    allow        (  false          (W18,W20)   -*-           )
}
```

Besides this terminological knowledge also a suite of behavioral knowledge is included constraining values of the terminology. In the example, we see a `behavior` block stating that yellow bikes are only allowed for large wheel sizes `W18` and `W20`. Also, in a table-based representation, we see that wheel support is only allowed for the small wheel sizes `W14` and `W16` and a drivers weight less than 30kg. The weight of the rider does not matter if no wheel supports are fitted and the wheel size is either W18 or W20. The product model specifies, that only small and lightweight kids should have a support wheel and the small wheels.

In addition to the elements mentioned above, the COOM language offers a variety of more refined expression options for mapping configuration knowledge (see [9]).

3.2. Applications of Product Configuration Profiles

When we look at commercial and research systems for product configuration, we realize that no system can map the full expressiveness of the COOM language. However, in most cases this is not necessary, as different industrial projects may have different requirements in terms of the expressive power. For example, simpler dependencies between configurable components may only necessitate the formation of basic relationships using core language elements. Hence, it is not essential for all implementations of reasoners to support every language feature. Consequently, more efficient reasoners can be crafted specifically for knowledge bases that utilize only certain language elements. The variability of expressiveness permits the development of various reasoners tailored to the different complexities inherent in knowledge elements.

We introduce *product configuration profiles* (profiles) in order to map these requirements and the features of the different system approaches. Each profile slices the language into a specific aspect, for instance the handling of numerical constraints or the reasoning with cardinalities. Partitioning the language into profiles offers a number of benefits:

1. **Knowledge Engineering:** When building a new product configuration knowledge base, the requirements can precisely be communicated with knowledge engineers, customers, and users.
2. **Engine Identification:** A product configuration engine working with the knowledge base has a clear requirements sheet. That way, selecting an appropriate engine will become a simpler task.
3. **Engine Comparison:** Alternative engines implementing the same profile can be evaluated and compared in an accurate manner, because a clear focus on specific profile aspects can be set.

It is worth noticing that we do not present an exhaustive set of profiles. Rather, we expect additional profile slices to appear with the growing number of industrial projects and community feedback. Furthermore, the profiles presented can be regarded as a general categorization of product configuration knowledge independent of COOM. The COOM language is only used here to visualize the profile properties.

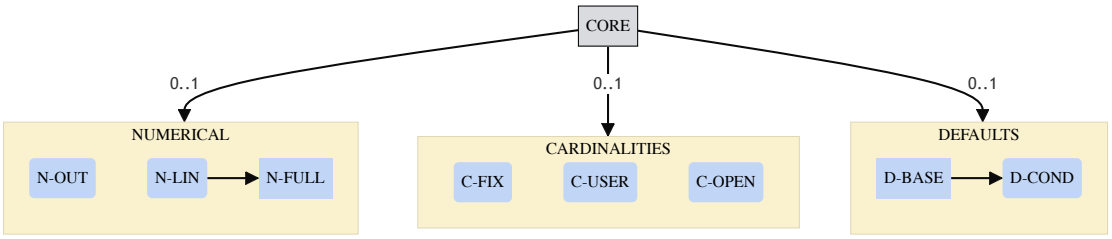

Figure 4: Profiles for product configuration.

Figure 4 depicts the current profiles as a hierarchical view, where the CORE profile (see Section 3.3) is a required profile and all other profiles are optional additions to this profile. Each profile definition refers to an example knowledge base representing a special kind of bike. The actual implementations of the bikes can be found in the appendix of this paper.

3.3. The CORE Profile

We introduce the **CORE** profile providing the essential language elements required to establish a basic configuration knowledge base. Products are described through *structures* and *enumerations* of predefined choice values. The relationships among the values of different features are established through constraints. These constraints specify combinations of choice values that are either permitted or prohibited. The CORE profile is required and all following profiles build upon it.

3.4. Handling of Numerical Constraints (N)

The **N-OUT** profile is designed for calculations of numerical values that do not restrict the solution space. Because the outcomes of these calculations do not have significance to whether some configuration is valid or not, they are considered external to the solution space. A practical use of the N-OUT profile might be calculating the price or weight of a product after the user has entered a valid configuration.

In contrast, the **N-LIN** profile facilitates calculations of numerical values within the solution space, similar to the following N-OUT profile. Then, the calculation results further constrain the range of valid configurations, typically occurring when computational outcomes are integrated within other constraints. In the N-LIN profile calculations in the solutions space are only allowed, if they are based on linear equations. A common application of N-LIN is summing a collection of numerical values. The use of non-linear numerical operators such as sqrt, cos, and pow will cause the system to exit this profile and will yield N-FULL.

The **N-FULL** profile extends N-LIN when handling numerical calculations in the solution space. Here, also superlinear calculations are allowed, whereas N-LIN only permits linear calculations. It is important to note that implementing the N-FULL profile can be challenging for declarative configuration engines, as they must find states for numerical values during the configuration process that satisfy all equations.

3.5. Handling of Cardinalities (C)

In the previous example, we illustrated that features can possess a cardinality, as seen when we defined two instances of the type `Bag` in the `PaperKidsBike` product.

The **C-FIX** profile supports the explicit specification of fixed cardinalities for a specific element (as done in the example). However, introducing cardinality ranges, such as 2..5, will exit this profile.

Conversely, the **C-USER** profile accommodates user-defined cardinalities for a specific element, allowing users to specify the number of instances during the configuration process. For example, we would require the C-USER profile, when the concrete amount of `Bag` instances is not defined in the knowledge base (as in the `PaperKidsBike`) but can be changed by the user during the configuration session.

Lastly, the **C-OPEN** profile permits the representation of arbitrary cardinality ranges for features. This flexibility means that the exact cardinality does not need to be predetermined in the knowledge base or by the user during configuration. Instead, the required number of instances is dynamically calculated by the reasoner, based on the constraints defined in the knowledge base. It is worth noticing, that—to the knowledge of the authors—only special purpose reasoners support open cardinalities in a smart product configuration setup. In the Bag example, the reasoner might derive the required number of `Bag` instances for a user-entered value such as the required total capacity for instance.

3.6. Handling of Defaults (D)

Typically, a configuration is created by users entering values for enumeration instances and primitive type instances. In addition, *default values* are recommendations for values that were not yet selected. A default value is automatically assigned to a value if the user has not entered a value previously, provided that this default value does not violate any existing constraints.

The **D-BASE** profile supports the establishment of simple default values. In this profile, default values are initially determined by the reasoner but may be overwritten by the user or through behavior knowledge. However, this profile only supports the basic assignment of default values, and no alterations to these defaults are permitted during the configuration session.

The **D-COND** profile expands upon D-BASE by allowing for the assignment of conditioned default values. These defaults can be limited to particular combinations of other values, meaning they may be adjusted during the configuration session if certain values are entered by the user or triggered by another constraint.

It is conceivable that a knowledge base might contain contradictory default knowledge, especially when multiple default rules that apply to the same feature overlap. The resolution of such conflicts typically depends on the reasoner's specific conflict resolution strategy.

4. Conclusions

This paper proposed an approach for partitioning knowledge for building configuration systems into profiles, based on different core aspects of product configuration. We introduced the open

language COOM and used it to describe a general set of profiles for product configuration knowledge. Clearly, with new profile requirements the language will also grow. This can for instance be helpful for reasoner selection. That is, if a reasoner proves very efficient for a particular product model, then for other product models having the same language profile, that particular reasoner is also an interesting choice.

The presented work is an on-going task. In the future, we aim to conduct a systematic comparison of current state-of-the-art reasoners in this field. This will involve evaluating our approach with respect to the adaptability across different engines. Additionally, we plan to explore the semantic interpretation of the COOM language by describing an RDF representation of the COOM syntax. This effort will facilitate a deeper understanding and broader implementation of the language. Finally, there is a clear need for more research into practical use cases, particularly concerning the extension of profiles. Investigating these areas will enable us to refine the COOM framework to better suit real-world applications. An interesting example will be the generalization of the default profiles to optimization of configurations.

References

[1] J. McDermott, R1: A rule-based configurer of computer systems, Artificial Intelligence 19 (1982) 39–88.

[2] F. Puppe, Systematic Introduction to Expert Systems, Springer, Berlin, 1993.

[3] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen, Knowledge-based Configuration: From Research to Business Cases, 1 ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.

[4] L. Hvam, N. Mortensen, J. Riis, Product Customization, Springer, 2008.

[5] A. Falkner, G. Friedrich, A. Haselböck, G. Schenner, H. Schreiner, Twenty-five years of successful application of constraint technologies at siemens, AI Magazine 37 (2017) 67–80. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/2688. doi:10.1609/aimag.v37i4.2688.

[6] J. Tiihonen, Characterization of configuration knowledge bases, in: Workshop on Intelligent Engineering Techniques for Knowledge Bases (IKBET), 2010.

[7] M. Krötzsch, Owl 2 profiles: An introduction to lightweight ontology languages, in: Reasoning Web Summer School 2012, Springer, 2012, pp. 112–183. doi:10.1007/978-3-642-33158-9_4.

[8] W3C OWL Working Group, OWL 2 Web Ontology Language Document Overview (Second Edition) - W3C Recommendation 11 December 2012, 2012. URL: http://www.w3.org/TR/owl2-overview/.

[9] coom-lang, https://www.coom-lang.org, 07.2024.

[10] coom-suite, https://github.com/potassco/coom-suite, 07.2024.

A. The Bike Family: Profile Examples in COOM

We illustrate each profile presented in this paper by a small demonstration implementation, each introducing a new kind of bike. Please note, that for more verbose introduction of the COOM syntax we refer to the language description [9].

A.1. COOM Core: KidsBike

```
product {
    KidsBike kidsBike
}

structure KidsBike {
    Color  color
    bool   supportWheel
    Wheel  wheelSize
}

enumeration Wheel { W14 W16 W18 W20 }

enumeration Color { Red Yellow }

behavior KidsBike {
    combinations (  supportWheel   wheelSize )
    allow        (  true           (W14,W16) )
    allow        (  false          (W18,W20) )
}

behavior KidsBike {
    condition color = Yellow
    require  wheelSize = W18 || wheelSize = W20
}
```

A.2. COOM N-OUT: Weighty Bike

The product `WeightyBike` focuses on the calculation of the `totalWeight` that is based on the weight of the `wheelSize` and a fixed weight of the frame (`frameWeight`). The computation of the `totalWeight` uses `wheelWeight` and `frameWeight`. The result is not used in any constraint, which makes it independent from the solution space of possible valid configurations.

```
product {
    WeightyBike weightyBike
}

structure WeightyBike {
    WheelSize wheelSize
    Weights   weights
}
```

```
9
10 structure Weights {
11     num /gr wheelWeight
12     num /gr frameWeight
13     num /gr totalWeight
14 }
15
16 enumeration WheelSize {
17     // Attribute definitions
18     attribute num/inch size
19     attribute num/gr   weight
20
21     // Choice values of the type
22     W14 = ( 14 , 550  )
23     W16 = ( 16 , 550  )
24     W18 = ( 18 , 600  )
25     W20 = ( 20 , 650  )
26     W22 = ( 22 , 700  )
27     W24 = ( 24 , 800  )
28     W26 = ( 26 , 900  )
29     W28 = ( 28 , 1000 )
30 }
31
32 behavior Weights {
33     imply frameWeight = 10000
34     imply wheelWeight = 2 * root.weightyBike.wheelSize.weight
35
36     imply totalWeight = frameWeight + wheelWeight
37 }
```

A.3. COOM N-LIN: Comfort Bike

The product `ComfortBike` offers front and rear bags, that come in different sizes and weights, respectively. Since this bike focuses on comfort, it also offers a suspension, that can be configured based on the expected weight of the bike. The total weight is calculated based on the weights of the rider and the bags. The value of the total weight is then used in a constraint stating that the possible weight of the suspension should be compatible with the expected total weight. Since, the computed value is based on a sum, the calculation is a linear equation.

```
1 product {
2     ComfortBike comfortBike
3 }
4
5 structure ComfortBike {
6     FrontBag frontBag
7     RearBag  rearBag
8     Suspension suspension
9     num /kg riderWeight
10    num /kg totalWeight
```

```
}

enumeration FrontBag {
    attribute num /kg maxWeight

    small   = (  5 )
    medium  = ( 10 )
    large   = ( 15 )
}

enumeration RearBag {
    attribute num /kg maxWeight

    small   = ( 10 )
    medium  = ( 15 )
    large   = ( 25 )
}

enumeration Suspension {
    attribute num /kg minWeight
    attribute num /kg maxWeight

    s1 = (   0  120 )
    s2 = ( 120  135 )
}

behavior ComfortBike {
    default riderWeight = 100
    imply totalWeight = (riderWeight + frontBag.maxWeight + rearBag.maxWeight)
}

behavior ComfortBike {
    require (totalWeight  > suspension.minWeight)
    require (totalWeight <= suspension.maxWeight)
}
```

A.4. COOM N-FULL: Sunday Bike

The product `SundayBike` is made for relaxing Sunday rides. It calculates the `effectiveTopTubeLength` that is based on the value of the `reach`, `stack`, and `seatTubeAngle`. The computation of the `seatTubeAngle` uses the `tan(seatTubeAngle)`, which makes it a non-linear computation.

Subsequently, a requirement exists, that the bag of the top tube needs to be smaller than the value of the `effectiveTopTubeLength`, i.e., `(topTubeBag.length + 30) < effectiveTopTubeLength`. This constraint influences the solution space of possible valid configurations.

```
product {
```

```
    SundayBike sundayBike
}

structure SundayBike {
    num /mm reach
    num /mm stack
    num /mm seatTubeAngle
    num /mm effectiveTopTubeLength
    TopTubeBag topTubeBag
}

enumeration TopTubeBag {
    attribute num /mm length

    short   = ( 550 )
    middle  = ( 600 )
    long    = ( 650 )
}

behavior SundayBike {
    default reach = 470
    default stack = 600
    default seatTubeAngle = 73
    imply effectiveTopTubeLength = reach + (stack / tan(seatTubeAngle))
}

behavior SundayBike {
    require (topTubeBag.length + 30) < effectiveTopTubeLength
}
```

A.5. COOM C-FIX: Shopping Bike

The product `ShoppingBike` needs to carry a lot of shopping items and therefore offers exactly two instances of the `Bag` type for the front and exactly three instances of the `Bag` type for the rear. The constraint states, that the sizes of the two front bags need to be equal to each other.

```
product {
    ShoppingBike shoppingBike
}

structure ShoppingBike {
    2 Bag frontBag
    3 Bag rearBag
}

structure Bag {
    Color color
    Size  size
}
```

```
14
15 enumeration Color { Green Blue Red }
16
17 enumeration Size {
18     small   = ( 10 12 )
19     medium  = ( 15 16 )
20     large   = ( 25 20 )
21 }
22
23 behavior ShoppingBike {
24     require frontBag[0].size = frontBag[1].size
25 }
```

A.6. COOM C-USER: City Bike

The product `CityBike` is designed to be very safe on the road and therefore offers an extensive configuration of spoke reflectors for the wheels. In principle, it is possible to configure at maximum 99 reflectors per wheel, but during the configuration process the user has to specify the concrete amount of instances.

```
1  product {
2      CityBike cityBike
3  }
4
5  structure CityBike {
6      Wheel frontWheel
7      Wheel rearWheel
8  }
9
10 structure Wheel {
11     0..99 SpokeReflectors spokeReflectors
12 }
13
14 structure SpokeReflectors {
15     ReflectorType type
16     Color color
17 }
18
19 enumeration ReflectorType { oval thin }
20
21 enumeration Color { Green Blue Red }
```

A.7. COOM C-OPEN: Cargo Bike

The `CargoBike` specializes in transporting luggage. The user enters the desired `totalVolume` and `totalWeight` of the items to be transported. Subsequently, the reasoner calculates the required number of Bag instances with their respective sizes.

In the knowledge behavior definition we see that the sum over all instances of `bags.size.maxWeight` is compared with the user entered value `totalWeight`. This also holds for `bags.size.volume` and `totalVolume`. Here, we compute the sum over all currently existing instances since no concrete index is given.

With this knowledge the reasoner must generate a sufficient number of `Bag` instances, each with its corresponding size, to ensure that the equations are satisfied. To guide the reasoner during this process of instance generation, we include the optimization statement `minimize countBags`. This directive aims to generate the least amount of bags that still satisfies `totalWeight` and `totalVolume`.

```
1  product {
2      CargoBike cargoBike
3  }
4
5  structure CargoBike {
6            num /kg totalWeight
7            num /l  totalVolume
8            num     countBags
9      0..99 Bag     bags
10 }
11
12 structure Bag {
13     Color color
14     Size  size
15 }
16
17 enumeration Color { Green Blue Red }
18
19 enumeration Size {
20     attribute num maxWeight
21     attribute num volume
22
23     small   = ( 10 12 )
24     medium  = ( 15 16 )
25     large   = ( 25 20 )
26 }
27
28 behavior CargoBike {
29     require sum(bags.size.volume) >= totalVolume
30     require sum(bags.size.maxWeight) >= totalWeight
31 }
32
33 behavior CargoBike {
34     imply countBags = count(bags)
35     minimize countBags
36 }
```

A.8. COOM D-BASE: White Bike

The `WhiteBike` allows for three different colors, but is assigned to `White` by default, if no other value is manually chosen. The user (or constraints) are able to set the color to a different value.

```
product {
    WhiteBike whiteBike
}

structure WhiteBike {
    Color color
}

enumeration Color { White Black Red }

behavior WhiteBike {
    default color = White
}
```

A.9. COOM D-COND: Poor Light Bike

The `PoorLightBike` automatically chooses the color `White` in case the user wants to configure the bike without lights. This way, the bike is at least a little more visible in the dark thanks to its color.

```
product {
    PoorLightBike poorLightBike
}

structure PoorLightBike {
    bool lights
    Color color
}

enumeration Color { White Black Red }

behavior PoorLightBike {
    condition lights = false
    default color = White
}
```