COSLING Configurator

CHARPENTIER Antoine and **FAGES Jean-Guillaume** and **LAPEGUE Tanguy**[1]

Abstract. This paper introduces COSLING Configurator, a solution for complex products and services modeling and configuration. Based on Constraint-Programming technologies through Choco Solver[1], COSLING Configurator handles a wide range of constraints and defines some of its own formalisms. Early feedbacks from industrial use cases have already validated the interest of COSLING Configurator.

1 Introduction

The creation from scratch of a domain-specific configurator has been studied and deemed an especially difficult task, as "poor decision-making in one phase may produce escalating negative consequences in the subsequent phases, until the configurator project eventually fails." [2], hence the need of generic and easy to use configurator.

COSLING Configurator is a SaaS suite designed around the creation and deployment of configuration forms. These are interactive forms accessible to any non-expert user, where every action triggers a constraint validation and propagation, to ensure that even when handling complex objects, every result of the form is correct. Its main application is the generation of quotes satisfying a set of business rules without requiring the user to know them. The target benefits of this tool are:

- Improving the sales experience by allowing more customization while reducing quoting time and offering a digital experience.
- Digitizing and centralizing expert knowledge so that it can be used easily by non experts and maintained over time.
- Building a configurator easily without programming skills, thanks to a code-less ergonomic interface, therefore saving the cost of developing and maintaining ad hoc quoting solutions.
- Allowing deeper integration with other Information Systems

The tool is collaborative: it enables the sharing and permission-control of resources and forms so that the administrator can enable internal or external users to view, interact with, or edit configuration models or forms. For instance, an expert can be the only actor with edition rights, with salespeople having run access. Or a team of experts could collaborate on the edition of a model, with the configurator being deployed publicly on a website.

COSLING Configurator has been successfully used for quote generation and design automation on industrial use cases (e.g. pumps, centrifugals, wood chip boilers, electric motors, ...).

As depicted by Figure 1, the suite consists of an editor for creation, storage and management of models, a configuration kernel and a configuration interface that together allow for immediate testing and first results off-the shelf. The non kernel modules can be replaced externally for advanced uses. The target workflow of the suite is as follows:

- Store business data
- Build standalone models
- Reuse models as components to represent larger systems
- Design the configuration user interface and workflow
- Deploy the configuration interface and make it accessible to authorized actors

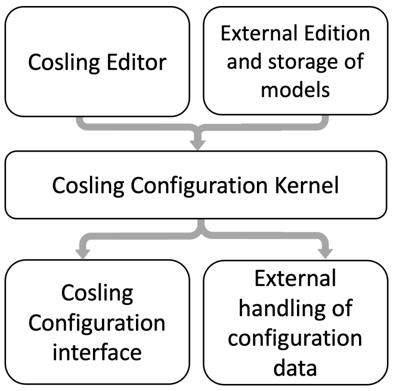

Cosling Editor
External Edition and storage of models
Cosling Configuration Kernel
Cosling Configuration interface
External handling of configuration data

Figure 1. Configurator architecture.

2 Related Work

In the era of modern industry, the number of the ways to answer a given need, whether it be in the form of material products, software [3], processes, or sets of requirements [4], has gone up exponentially. Thus, the process of setting up a single solution has become increasingly heavy [5], to the point of necessitating a class of digital solutions dedicated to partly automate that process: configurators.

Configurators are not a new concept [6], nor are they becoming obsolete [7]; their implementation has shown substantial benefits in several areas, from sales to industrial problem solving [8]. The abstract nature of configuration raises the possibility of unifying configurator development with a generic solution. Such a solution is relevant, as the cost and risk of failure of developing a domain specific configurator are high [2]. Several generic configuration solutions have been developed in the past, ranging from research driven engines to integrated modules of CRM suites [9]. Motivated by the

[1] COSLING S.A.S., France, email: contact@cosling.com, website: https://www.cosling.com/

inherent constraint based search aspect of configuration that overlaps with its core expertise, COSLING introduces a configuration solution centered around user experience, high flexibility in modeling and integration, and performance. It is built on the belief that multistage[10] visual configuration [11] has the potential to bring tangible gains with reasonable effort.

3 Configuration Paradigm

This section will present what constitutes a configuration model in the COSLING Configurator paradigm.

3.1 Attributes

Attributes have an initial *domain*, and their value in a final solution will be determined either by user actions or through the propagation of constraints by the engine, as a consequence of user actions. They can be seen as variables in the constraint programming paradigm. Constraints bind attributes together so that their values are always coherent with each other.

We can allow or disallow values for a numerical attribute when declaring its domain. Indeed while "integer" or "floating" are shorthands for the maximum possible range for both of these types, the general form of a numerical domain is a list of inclusive ranges. Thus, we can declare an attribute with a domain of [0,10], or a list or ranges (including a singular one) [2.5,3.5], [4.0,4.0], [4.5,5.5]

A specific case is the *$REFERENCE* variable which serves as a selector for references of a catalog, or variables with an *Enumeration* type . Although their value is represented by an integer variable internally, their value always appears as the label of the corresponding references. Both concepts are presented below.

For instance, if we want to configure a quantity of products costing 2.5€ to buy, we will use a positive integer attribute for the quantity and a floating point attribute for the price, bound by the simple constraint $price = quantity * 2.5$. The user could set the quantity and see the price, or set a target price and let the engine search for a matching quantity of products.

3.2 Enumerations

Enumeration are a simple utility concept meant to give names to integer values. Enumerations have several purposes: centralizing recurring values across the application and illustrating basic choices in modeling and configuration.
$\{false : 0, true : 1\}$ or $\{blue : 0, white : 1, red : 2\}$ are instances of enumerations that can be used for providing a choice in the model.
$\{standard\ diameter\ 1 : 125,\ standard\ diameter\ 2 : 165\}$ is an instance of enumeration that can be used for both choices and computations. As mentioned above, the usage of an enumeration is as the domain of an attribute.

3.3 Catalogs

The catalog is the main way to store data about the object to be configured. A catalog is comparable to a database table in that it represents many given versions of a class of object.

If the object to be configured includes a metal frame for instance, a simplified catalog might look like this:

	Price	Color	Weight
	int	Enumeration: color	[100,500]
Frame A	100	White;Blue	150
Frame B	200	Red	175..300
Frame C	300	Blue	225

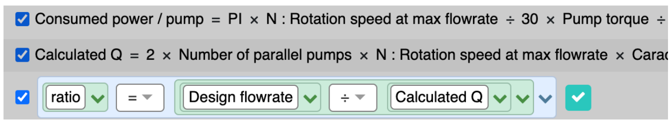
Table 1. Catalog definition example.

That is to say, a catalog defines attributes (columns) with their name and domain, then a collection of lines referred to as "references" in the paradigm, specifying a specific value for every attribute. Such domain may consist of a single value or include some variability, as may be seen in Table 1. A catalog reference can also include an image, which will be rendered in the configuration view and generated quotes.

Note that it is possible to design entirely a model through a catalog. As they are quite similar with Excel format, Catalogs form a simple and effective way to model products.

3.4 Constraint expression

Constraints are expressed with two abstract element types: constraints and terms. A term results in one or multiple variables, and constraints bind these variables together. Rather than implementing a DSL for the constraint declaration, it has been decided to use a visual interface relying on term and constraint selectors so as to guide the user in the edition process:



✓ Consumed power / pump = PI × N : Rotation speed at max flowrate ÷ 30 × Pump torque ÷

✓ Calculated Q = 2 × Number of parallel pumps × N : Rotation speed at max flowrate × Carac

✓ ratio = Design flowrate ÷ Calculated Q

Figure 2. Constraint edition.

3.4.1 Terms

- Paths. This is a dot separated relative path to an attribute. There is no "parent" keyword by design, so a path only goes down the tree, never up. NB: if one or many sub-models included in the path have more than one instance, a path will return several variables. e.g. $frame.\$REFERENCE$ (one variable), $Wheels.price$ (several variables, cf **Figure 1**)
- All. This will return every attribute with the given name from the constraint's model or its sub models, with infinite depth. e.g. $All(price)$
- Constants. They are expressed as a string, which can either represent an integer, a float, or the label of a catalog reference. e.g. $1, 1.3, FrameA$
- Unary expressions. $sin(angle), round(price)$
- Binary expressions. $discount * catalog\_price$
- Ternary expressions. e.g. $if(discounted == true)\ then\ price * (1 - discount)\ else\ price$
- N-ary expressions. Mathematical expression involving n terms. The expression will take into account every variable returned by every term. Sum, max, min... e.g. $sum(all(price), frame.discounted\_price)$

3.4.2 Constraints

- Arithmetic. A mathematical relation between two terms. e.g. $discounted = true$, $load >= 200$

- N-ary. A mathematical relation between n terms. Will take into account every variable returned by every term. Equal/different. e.g. $equal(all(axle\_type), chosen\_type)$
- Logical. Binds two constraints by a logical connector. If/Or/Xor/And/IfOnlyIf, e.g. $IfOnlyIf((homogeneous = true), equal(all(axle\_type))$

3.5 Models

A model is a tree structure where each node has attributes, constraints, at most one catalog, and (sub) models (child nodes). The arborescent nature of models has several purposes:

- Organizing attributes in large models and making the edition of individual parts simpler. Indeed constraints are defined in models, and can include the attributes of child models, but not parent models.
- Reusing standalone models in several, larger models. It is possible to encapsulate the logic of one object reused across several projects in a standalone model, then include it where needed. If for instance we have a pair of components that are always used together and bound by the same constraints, we can create the corresponding model as standalone to avoid having to write that logic again everywhere the pair is used.
- Repeating a sub model a dynamic number of times. It is possible to bind an integer defined in the parent of a model to it's number of instances. This creates a specific behaviour at the configuration run-time: the corresponding sub models will be created upon the instantiation of the instance number variable. In other words, they may be created and destroyed multiple times during configuration depending on changes of the variable's domain.
- Importing several catalogs in the same standalone model through child nodes. As of now, multiple catalog is inheritance unsupported. Not only would it require dealing with attribute conflicts and multiple reference variables, we also believe that encouraging the Composition Over Inheritance [12] principle in the configuration paradigm leads to overall cleaner model designs, therefore reducing long term user frustration.

A model has several ways of declaring attributes: importing a (sub) model, a catalog, or defining custom attributes.

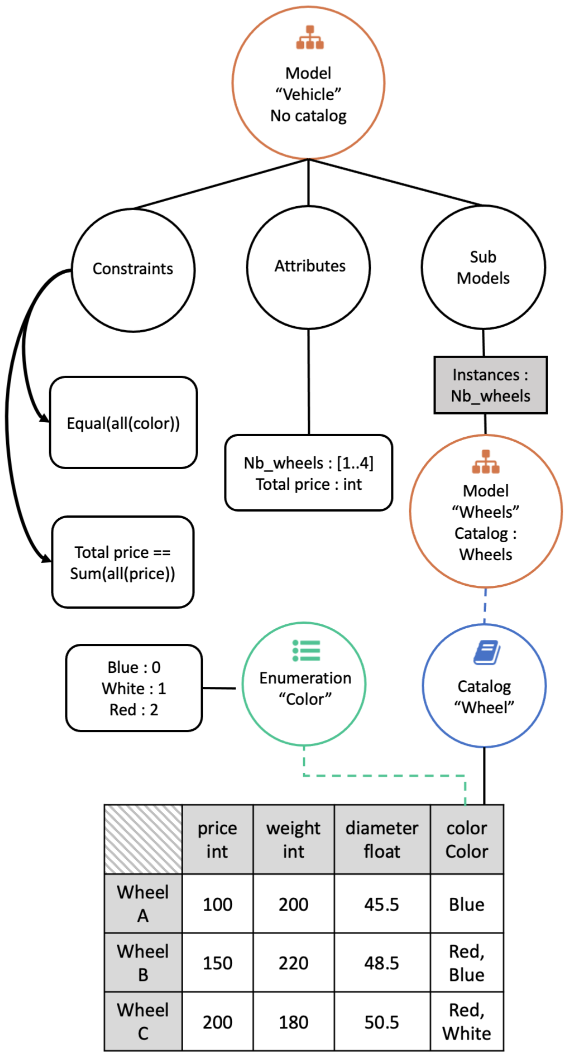
Importing a catalog adds every attribute from it to the model, as well as a $REFERENCE attribute. $REFERENCE is linked by an implicit *element* constraint to the catalog attributes, so that selecting a reference restricts every attribute to the value of the corresponding line in the catalog, and conversely, that setting an attribute restricts $REFERENCE to only keep the references that match this value.

Custom attributes can be added freely to any model, together with a set of constraints to define their behavior. As such, custom attributes are used to complement the information stored into catalogs. For instance, applying a discount should be independent of the chosen catalog. We would therefore define a custom floating point attribute and bind it to the catalog price by the constraint $discounted\_price = (1 - discount) * catalog\_price$". Importing a catalog and defining custom attributes are not mutually exclusive.

Figure 3 represents a basic model to help understanding the structure and dependencies of a model.

3.6 Forms

Although it is possible to configure/run a model directly, exposing every attribute to the user in generic way may be confusing for the user and quite counter productive for sales.



▨	price int	weight int	diameter float	color Color
Wheel A	100	200	45.5	Blue
Wheel B	150	220	48.5	Red, Blue
Wheel C	200	180	50.5	Red, White

Figure 3. Basic example.

For this reason, we propose to configure the configuration view through Forms. A Form describes the configuration interface of a model. If the model can be seen as the back-end of configuration, the Form can be seen as its front-end counterpart.

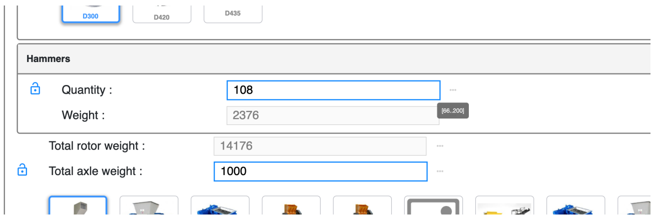
A form is defined as a collection of steps. A step serves as a "strong" delimiter for the configuration workflow. In most cases, choices made in a step by the user are final and may be used as a basis for the following steps.

Steps are constituted of fields organized in groups. It is possible to nest groups as needed.

A field is simply the association of a label and a path. This path is the same type of path used in constraints, starting from the root model. If resolving the path results in multiple attributes (in the case of variable instances), there will be one field for every instance. Lastly, there are several interface options for a given field: a plain text box, a select for domains that can be enumerated (reasonably sized integer or named domains), a checkbox with two values. A field can also be set as read-only, for when the displayed attribute is not meant to be configurable but only as a result of other choices, or when the choice has already been made in an earlier step.

4 Configuration Process

Running a form in the editor produces such a view:



Hammers	
Quantity :	108
Weight :	2376
Total rotor weight :	14176
Total axle weight :	1000

Figure 4. Sample configuration view

For each field, the following information are displayed:

- A current value. Fields always contain a value, whether specified by hand or computed by the solver. These values are compliant with constraints, thus the form shows a ”valid” solution from the start.
- Whether or not it has been manually locked by the user is represented by a lock that is either open or closed. Locking a field means that the engine will now only search for solutions where this field's value does not change. Any non locked field is subject to change when locking another field, so that the displayed value is still valid afterwards.
- The current domain, available by hovering the icon to the right of every field. The domain is the range of values still coherent with constraints and previous user choices. The domain of a locked field is restricted to one value.

Attempting to give a value outside of its domain to a field initiates a conflict resolution process outputting a list of choices to unlock for the desired commit to be feasible.

This interface displays information received from the configuration API. When using the configuration as a micro-service, the same parameters (value, domain, locked state, etc.) should be handled by a third party GUI.

Upon reaching the last step, the configured result is available visually through the form, as a word document generated from the data, and can be pushed over https as JSON to the host company's server for any custom treatments. Configurations could be saved for later in a customer space or an order could be directly placed, for instance. In most cases, this is outside of the configurator's scope of responsibility.

5 Conclusion

We have introduced COSLING Configurator, a web platform to design and run configuration models in order to digitize sales and design automation. It is based on a simple paradigm aiming at helping business data remain as structured and clear as possible. Models can be generated through an ergonomic interface or programmatically. Based on the same model, different configuration forms can be designed and deployed so as to to be used as part of any flow that involves configuration. The core value proposition lies in the simplicity of the paradigm and the strength of the configuration kernel. Finally, the solution can be integrated as a configuration micro-service through web APIs in order to benefit from its inference power in a wider system. Thus, the COSLING Configurator is a generic enough solution to adapt to a wide range of needs, but domain specific enough to not define the architecture of users too heavily. Early feedbacks from industrial use cases have validated the strength of the Kernel and the interest of our code-less user interface.

Since Configuration problems are usually under constrained, picking a good solution from all valid solutions is an important task. Therefore, future work will mainly focus on making it possible to specify what a good solution is (by handling optimization criteria) and how to build it (by handling search strategies). So as to make COSLING Configurator even more interactive, we also plan to improve our conflict resolution by giving several alternatives to solve conflicts. By collecting feedback from our users, we are also looking to further improve user experience, especially regarding table edition and model refactoring which are key points. Lastly, efforts will be made to facilitate the configurator integration within other systems, such as ERPs, CRMs and CAD environments.

Academic licences for Research and Teaching can be granted on demand.

REFERENCES

[1] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco Solver Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[2] Anders Haug, Sara Shafiee, and Lars Hvam. The causes of product configuration project failure. Computers in Industry, 108:121–131, 06 2019.

[3] F. Liguori and F.A. Schreiber. The software configurator : an aid to the industrial production of software. pages 487–492, 1978.

[4] Michel Aldanondo and Élise Vareilles. Configuration for mass customization: How to extend product configuration towards requirements and process configuration. Journal of Intelligent Manufacturing, 19:521–535, 10 2008.

[5] David Mick, Susan Broniarczyk, and Jonathan Haidt. Choose, choose, choose, choose, choose, choose, choose: Emerging and prospective research on the deleterious effects of living in consumer hyperchoice. Journal of Business Ethics, 52:207–211, 01 2004.

[6] F. Beuger, T. W. Sidle, L. W. Leyking, and A. G. Livitsanos. A programmable configurator. In Proceedings of the 11th Design Automation Workshop, DAC '74, page 177–185. IEEE Press, 1974.

[7] Yue Wang, Wenlong Zhao, and Wayne Xinwei Wan. Needs-based product configurator design for mass customization using hierarchical attention network. IEEE Transactions on Automation Science and Engineering, 18(1):195–204, 2021.

[8] Anders Haug, Lars Hvam, and Niels Henrik Mortensen. The impact of product configurators on lead times in engineering-oriented companies. AI EDAM, 25:197–206, 05 2011.

[9] R. Sabin, D.; Weigel. Product configuration frameworks-a survey. IEEE Intelligent Systems and their Applications, 13, 1998.

[10] Jeppe Bredahl Rasmussen, Anders Haug, Sara Shafiee, Lars Hvam, Niels Henrik Mortensen, and Anna Myrodia. The costs and benefits of multistage configuration: A framework and case study. Computers Industrial Engineering, 153:107095, 2021.

[11] Lars Hvam and Klaes Ladeby. An approach for the development of visual configuration systems. Computers Industrial Engineering, 53(3):401–419, 2007.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1 edition, 1994.