Product Customization as Linked Data

Edouard Chevalier and François-Paul Servant

Renault SA
13 avenue Paul Langevin
92359 Plessis Robinson, France
{edouard.chevalier,francois-paul.servant}@renault.com

Abstract. Ranges of customizable products are huge and complex, because of the number of features and options a customer can choose from, and the many constraints that exist between them. It could hinder the publishing of customizable product data on the web of e-business data, because constraints are not tractable by agents lacking reasoning capabilities. But the configuration process, which helps a customer to make her choice, one step at a time, is a traversal of a graph of partially defined products - that is, Linked Data. Reasoning being hosted on the server, its complexity is hidden from clients. This results in a generic configuration API, in use at Renault. As configurations can be completed to valid commercial offers, the corresponding ontology fits nicely with GoodRelations. Benefits in e-business related use cases are presented: sharing configurations between media, devices and applications, range comparison based on customer's interests, ads, SEO.

Keywords: configuration, customizable product, Linked Data, GoodRelations, automotive.

1 Introduction and Motivation

Publishing product data to improve e-business performance and visibility on the web through Search Engine Optimization (SEO) is gaining momentum, thanks to vocabularies such as GoodRelations [1], and the Schema.org[1] initiative.

An increasing number of manufacturers and vendors have been trying to make their websites usable by agents and client applications, and accurately searchable by search engines. This has produced interesting results with products such as books or music, for instance: search results listing actual products - not web pages - and including links to commercial offers with price, ratings, etc.

But suppose you are looking for, say, a small car, with a gasoline engine, a sun-roof, air conditioning, and an adapter to connect your MP3 player: how nice if you could just type that in your search engine. Ah, and you're concerned with price, and you would like to compare CO2 emissions; well, your search engine probably won't help there. You might find some dedicated sites to provide comparisons between cars, but nothing precise enough to allow the exact kind

[1] http://schema.org/

E. Simperl et al. (Eds.): ESWC 2012, LNCS 7295, pp. 603–617, 2012.

of comparison you want: price and CO2 emissions of small gasoline cars with sun-roof, etc.

Hardly surprising: the data such applications require is not available yet. Are manufacturers reluctant to open up their data about their ranges? If they are, they will have to change as more daring competitors enter the game and begin publishing their data: the cost of not appearing at all in the results of searches made by potential customers would be overwhelming, particularly if such search results easily convert to purchase orders.

On the other hand, it should be noted that cars are more complicated to handle than books: books are searched on the basis of a very small set of properties (title, author,...); they are well identified (e.g. through ISBN); and comparisons between commercial offers only involve completely defined products. Whereas cars are customizable, a crucial aspect of the problem: rather than fully specified products, what you compare are sets of them, that is, partially defined products.

In industries practicing "Build to Order" of fully customizable products, ranges are huge, because of the number of features and options a customer can choose from: more than 10^{20} different cars are for sale at Renault, and 30 to 40 decisions are needed to completely differentiate one from them. Those ranges are not only huge, they are also complex, because of the many constraints between features which invalidate some of their combinations: if every combination of distinctive features and options were possible, there would be 10^{25} different Renault cars, not our mere 10^{20} - meaning you have only one chance upon 100,000 to define an existing Renault car, if you choose its specifications without taking the constraints into account.

Specifying those product ranges requires the use of a vocabulary able to represent the constraints. This can be done by means of Semantic Web languages [2], but using this data in practical applications requires sophisticated automatic reasoning to handle the constraints. Publishing such range definitions on the web clearly won't bring many practical results soon, as one cannot expect strong reasoning capabilities from client agents.

Though difficult to specify and hard to manipulate, these product ranges are nevertheless described rather effectively, for human users, by means of dedicated web applications called configurators. A configurator helps a user interactively define a product step by step, each step describing a valid partially defined product (PDP), with a start price and a list of remaining choices given all previous selections. Each of these choices links to another PDP until completion. Thus, the configuration process traverses a graph whose nodes are PDPs. Now identify each PDP with a URI returning the list of the PDPs it is linked to, among other relevant information: what you have is a description of the range as Linked Data. This is how a configuration engine can publish descriptions of complex products on the web of data, which agents without reasoning capabilities can effectively understand.

This should be of some benefit to any configurator application, whatever its actual implementation. To name but one - the easy sharing of PDPs between

applications, devices, and media such as social networks is an effective way by itself to increase visibility on the web.

Renault began to publish data about its range in this way, after implementing a Linked Data based configuration API on top of its configuration engine, a deductively complete reasoner allowing configuration in free order. Any valid combination of choices can therefore be handled as a PDP, and be published on the web of data. Agents can easily crawl this data.

This document is structured as follows. We begin with an overview of the configuration process. In section 3, we show how it can be modeled as Linked Data which provides for an easy to use configuration API. We describe its implementation and use at Renault in section 4. A GoodRelations compliant configuration ontology is proposed in section 5. Finally we discuss the benefits of the solution: a clean system architecture, reduced development costs for client applications, improved sharing of configuration-related information. This opens the way to novel applications, including e-business related ones.

2 Product Range Specification and Configuration

2.1 Product Range Specification

Because the set of different products that a customer can specify and order is too huge to be enumerated, ranges of customizable products are defined in intention.

The specification of a family of similar products (typically those of the same "model") is based on a "lexicon", i.e., a set of variables representing the relevant descriptive attributes: body type, type of fuel, color, etc. In a completely defined product, any of these variables is assigned one value and one only. Such a value is called a "specification" in ISO-10303 STEP AP 214 terminology, a term that we'll use throughout this paper. In the Renault range, the variables are discrete: any of them, e.g. the type of fuel, has a finite list of possible values, e.g. gasoline, diesel, electric, gasoline-electric hybrid.

Then a set of constraints restricts the possible combinations of specifications.

The Product Range Specification (PRS) is therefore a Constraint Satisfaction Problem (CSP), and the many PRS related questions that have to be answered in the day to day operation of business are computationally hard (SAT being NP-complete). Renault has developed tools, based on a compiled representation of this CSP. The computationally hard part of the problem is fully solved in an offline phase, guaranteeing bounded and fast response times for most of the queries: for configuration queries, time is linear on the size of the compiled representation, which happens to remain small enough [3].

2.2 Presenting a Range of Customizable Products to Customers

A configurator application is the main way of presenting a complex range of customizable products to customers. It is a decision support tool that guides users to desirable - and valid - product configurations. Most web sites of automotive

constructors give access to a configurator application, often through a link entitled "build your own car" - although it will not actually be built, but chosen from a huge set.

Most of the configurator applications help a user interactively define a product step by step, each step describing a valid partially defined product, in the sense that it can be completed, without changing any of the current selections, into an existing fully specified product, which can be ordered. We'll call "Configuration" any such valid, partially defined product: in other words, any state of the configuration process.

Note: A configurator application may conceivably have to handle "invalid configurations", that is, combinations of specifications that are impossible. This can happen, for instance, if the user is allowed to begin the configuration process by choosing features without any control of their compatibility; or if she is allowed to choose a feature incompatible with her previous selections. In this case, it is the responsibility of the configurator application to restore the consistency of the configuration, necessarily by excluding some of the previous user selections. For us, the word "Configuration" excludes such invalid combinations.

2.3 Features of a Configuration Engine

We list here the features that we think are necessary in a good configurator. Not all of them may be available in the implementations we see on the web, either because the software supporting the configuration process (the configuration engine) is not able to provide them, or because of a poor application design, which tends to stick to the old way of selling products, typically imposing a predefined order on the user to make her choices, which simplified the handling of the configuration process, at the expense of user comfort.

The main point is that the configuration engine should guarantee completeness of inference, that is, every consequence logically entailed by a given state in the configuration process gets actually inferred when in that state, and not later [3]. This is absolutely necessary if users are to freely choose from specifications compatible with their previous selections, and to be barred from making choices no valid configuration satisfies; in other words, if the whole range is to be made accessible to them, without their ever having to backtrack from dead-ends.

Here a list of desirable features in a functional perspective:

- free choice order
- pricing information, possibility to filter by a max price
- negative choices (configuring by excluding specification)
- permissiveness and conflict resolution: users are allowed to change their minds about previous selections, or to select a specification excluded by them (in the latter case, the system should advise on which choices to change in order to restore a valid configuration).
- completion: providing a completely defined product matching the configuration.

Configuration of the Model. Frequently, each of the distinct models in a given product range is described with its own lexicon. This imposes the choice of a model as the first step in the detailed configuration of the product.This required first step is a nuisance - a customer may be hesitating between two similar models, and would need more information to make her choice. As a partial remedy, we can devise a set of variables shared by all the models, e. g. the body and engine type, the CO2 emission level, etc.

Another way to alleviate this problem is by allowing textual searches in the whole union set of specifications over the different models, with the configuration engine checking whether the conjunction of the found specifications matches existing configurations (see how-to in 3.3).

2.4 Related Work

Product configuration is an open and active field of research, with an important community and lots of publications. But this paper is not about reasoning and the way it is implemented. In fact, one of the main points is precisely about hiding the complexity of reasoning from clients.

The exact context of this work is the borderline where product configuration meets e-Commerce applications of Linked Data. The main contribution in the same field that we know of is Volkswagen's "Car Option Ontology"[2]. Their approach is different: they publish the constraints, in a proprietary vocabulary. We think that such data cannot be effectively used by simple agents, because they do not have reasoning capabilities. Instead, we host the reasoning on the server, and free clients from the burden, ensuring maximal usability of the data that we provide.

3 Configuration API

The configuration process can be modeled as Linked Data. This provides the basis for a simple, yet generic, Configuration API.

3.1 Configuration as Linked Data

In a typical configuration application, the user is presented with successive choices in a way that she cannot choose incompatible specifications. Each configuration, that is each successive state of the configuration process, is characterized by the specifications selected so far. A configuration engine is responsible for ensuring that only valid choices are presented at each step.

Such an application can therefore be implemented as a GUI over a REST service which takes the description of a valid configuration (the list of the specifications already selected) as input parameter, something like :

```
configService?chosenSpec=spec1&chosenSpec=spec2&...        (1)
```

[2] http://purl.org/coo/ns

and returns the next list of specifications to be chosen from, all guaranteed to be compatible with the input. Choosing one of them is then just a matter of adding it to the list of the "chosenSpec" query parameters and of getting the updated state of the configuration process.

Note that a query such as (1) identifies a configuration, and can be used as a URI for the configuration in question; or, more precisely, redirect to an actual URI of it; therefore, we can improve the service by making it return the URI of the linked configuration along with each compatible specification: the representation of the configuration resource then contains a list of couples (compatible specification, linked Configuration).

Such a service makes it easy to implement a configurator application: accessing a configuration URI returns the data needed to build the corresponding web page: basically a list of links to the next configurations. Every configurator application on the web could be (re-)implemented this way: it is just a matter of wrapping the configuration engine in a REST service that provides the data needed to generate the HTML

When implementing such an application, play with HTTP content negotiation, in quite classical Linked Data style, to respond either with data or with a HTML page to a given configuration URI; either a page built from the data, or the unadulterated data themselves. In the HTML page, include the data as RDFa or microdata markup; stating in particular that the page describes the Configuration.

3.2 Querying

The Linked Data based API allows to crawl the range, starting from the root of the service (the "empty configuration"). Only valid configurations are returned.

It is useful to also provide a way to query the dataset. The template of the service (1) can be used as a simple querying API. Mind however that any combination of specifications may not be valid. The service should detect such invalid conjunctions and return a 404 Not found HTTP error. Only configuration engines that support free order can provide such functionality in every circumstance.

It would be tempting to query the dataset using SPARQL syntax:

```
SELECT ?conf WHERE { ?conf :chosenSpec :spec1, :spec2 . }
```

but according to SPARQL semantics, this should return all the configurations with spec1 and spec2 - possibly several billions of them. Instead we would not expect a list of configurations here, but one only - or zero if spec1 and spec2 are not compatible. It is feasible to implement the intended semantics with SPARQL, but the syntax is a bit cumbersome, therefore far less attractive. We therefore didn't implement a SPARQL endpoint.

3.3 Use Case: Implementing Text Search

With these two services, we can implement text based searching in the configuration space. Here is a sketch of a solution, assuming the configuration engine

supports free choice order. The configuration corresponding to a car model links to the specifications compatible with that model; now, using a text search engine tool such as Lucene, index the (model, specification) pairs with the text form of the specification as the index key. Then, searching for "air conditioning, sun roof, MP3" (say) will get you a list of (model, specification) pairs; making the configuration service conjoin the car model and relevant specifications will get you configurations matching your text search. This adapts to the case where the configuration engine only supports free order only after some main choices have been made; for instance, if choices for car model, engine and level of equipment are required before allowing all options to be chosen in free order.

4 Renault Implementation

Traditionally, we have been providing access to the functionalities of our configuration engine through a java API. Recent plans for important changes in the Renault web site sparked an opportunity to provide a Linked Data based access.

The definition of the current commercial offer is managed by upstream systems, then compiled into the binary data used by the configuration engine (size: <100MB). It is published by means of a REST service, such as described above: Linked Data is materialized on the fly when PDPs are queried (30 KB per PDP). When the definition of the range is updated, part of the knowledge base used by the configuration engine is replaced. URIs of PDPs include the release number of the knowledge base, so all previous URIs are "deprecated" - but they still can be queried by clients: an HTTP 301 redirects to the new URI, if the PDP still exists in the range. A 404 is returned otherwise. In the latter case, the service can be re-queried to get a "similar" product.

The implementation of the service uses Jersey[3] (the reference implementation of JAX-RS[4]). As of this writing (February 2012) only JSON data are returned, and only for German and Italian markets.[5] All functionalities of our configuration engine are made accessible through the JSON data, including querying in free order, maximum price, conflict resolution, completion, etc., (optional query parameters being added to the configuration URIs to implement some of them).

A cursory look at the data[5] may convey the impression a configuration URI does not contain the list of chosen specifications which defines the configuration. It does, though; only encoded in a short form. For we anticipated configurations would be shared on Twitter; using an URL shortener or an internal index might bring down performance as vast numbers of configuration URIs are generated: 100-300 to represent but one configuration. Indeed, many links are included since we provide free order of choices.

Regarding performances, the HTTP response time for accessing one configuration is around 20-30 milliseconds.

[3] http://jersey.java.net/
[4] http://jcp.org/en/jsr/detail?id=311
[5] http://co.rplug.renault.{de,it}/docs

5 Configuration Ontology

This simple ontology[6] describes the classes and properties involved in the modeling of the configuration process as Linked Data.

As a partially defined product whose completion to a valid product always exists, a configuration can be seamlessly described in the GoodRelations ontology framework and can participate in the web of data for e-business. This ontology is generic, that is, applicable to any kind of customizable product: it does not depend on the set of variables and specifications with which a given product is defined.

Examples. In the following, we use examples about a very simple range of cars, all of the same model called Model1. The lexicon contains four variables:

- Fuel Type: {Gasoline, Diesel}
- Temperature Control: {Heating, AirCond}
- Radio Type: {NoAudio, SimpleRadio, RadioMP3}
- Roof: {NormalRoof, SunRoof}.

The product range specification is defined in Tab. 1, by the list of specifications available on the three levels of equipment. The total number of different completely defined cars is 8.

Table 1. Product Range Specification example

	Fuel Type	Temperature Control	Radio Type	Roof
Low-end	{Gasoline,Diesel}	{Heating}	{NoRadio}	{NormalRoof}
Mid-range	{Gasoline,Diesel}	{AirCond}	{SimpleRadio}	{NormalRoof}
High-end	{Gasoline,Diesel}	{AirCond}	{RadioMP3}	{NormalRoof,SunRoof}

Notations. The RDF examples are written in Turtle syntax, using the prefix "co" for this configuration ontology, "gr" for GoodRelations, "vso" for the Vehicle Sales Ontology and "r" for the specifications.

5.1 Main Classes

co:Specification. Specifications are first class objects, identified by URIs. This is very natural in most cases, as the specifications correspond to "real world objects": a fuel type, a radio system, etc. This can also be handy in cases where literal values would appear to be enough at first sight; e. g., where variables are given values from physical ranges, such as widths, or CO2 emission levels. In such cases, modeling them requires more information than their mere basic type; e.g. a unit.

In GoodRelations parlance, a specification is an instance of gr:QualitativeValue (vso:FeatureValue in the case of vehicles).

[6] http://purl.org/configurationontology

co:Configuration. This is the main class, of course. A Configuration is a state in the configuration process. It is defined by a list of choices: cf. the co:definingChoice property. As explained in section 3, implementing the configuration process as Linked Data is based on listing the specifications compatible with a given Configuration, along with the configurations they link to. This is modeled through the co:ConfigurationLink class, and the co:possible property.

5.2 Definition of a Configuration

A Configuration is defined by the choices the user made (and the definition of the range): primarily, the selection of specifications. Other kind of choices, not directly involving specifications, may be allowed by the configuration engine: for instance, a user can set a maximal price ("a car that costs less than 10.000 euros"), or a maximal delivery time ("a car that I can get within one month").

co:definingChoice. Parent to all properties specifying the choices that define the Configuration: a Configuration is defined by the list of triples it is the subject of, and which have a co:definingChoice as their predicates.

co:chosenSpec. A SubProperty of co:definingChoice listing the specifications selected by the user:

```
ex:Conf1 a co:Configuration ;
    co:chosenSpec r:Model1, r:SimpleRadio.
```

Now say you want a radio, but you do not care what kind it is. Because a configuration engine may support choices such as r:SimpleRadio OR r:RadioMP3, if two or more of the co:chosenSpec of a Configuration correspond to the same variable, by convention they are to be interpreted as ORed (even XORed, by the way).

```
ex:Conf2 a co:Configuration ;
    co:chosenSpec r:Model1, r:SimpleRadio, r:RadioMP3.
```

This means that the car has either a r:SimpleRadio, or a r:RadioMP3, not both.

Choice order. Choices are made one at a time and in a given order, which may matter. Of course it doesn't impact the characteristics of the product in any way, but it can be used by the application, for instance to display a textual description of the configuration. This could be achieved with an additional co:choiceSeq property having rdf:Seq as its range.

co:maxPriceChoice. A subProperty of co:definingChoice, an upper limit set on the price of the configuration.

5.3 Description of a Configuration

Given the co:definingChoice(s) of a Configuration, some specifications are implied (included in any completely defined product matching the configuration),

some are impossible (they can no more be chosen), others are simply compatible: they still can be chosen among several alternatives. Given the co:definingChoice(s) of a Configuration, some specifications are implied, i. e., included in any completely defined product matching the configuration, some are impossible, i. e. they can no more be chosen, others are simply compatible: they can still be chosen from among several alternatives.

co:impliedSpec. Given the constraints between the specifications of our range, r:AirCond is implied on ex:Conf1:

```
ex:Conf1 co:impliedSpec r:AirCond.
```

co:possible and co:ConfigurationLink. On ex:Conf1, we still can choose the fuel type:

```
ex:Conf1 co:possible
    [ a co:ConfigurationLink ;
        co:specToBeAdded r:Diesel ;
        co:linkedConf ex:Conf1PlusDiesel .],
    [ a co:ConfigurationLink ;
        co:specToBeAdded r:Gasoline ;
        co:linkedConf ex:Conf1PlusGasoline .].
```

Here is one of the linked configurations:

```
ex:Conf1PlusDiesel  a co:Configuration ;
    co:chosenSpec r:Model1, r:SimpleRadio, r:Diesel.
```

HTML display of a co:ConfigurationLink: it corresponds to an hypertext link, whose href is the value of the co:linkedConf property. As for the text of this link, the rdfs:label of the co:specToBeAdded value is quite adequate. It can be directly included in the RDF as a rdfs:label of the co: ConfigurationLink:

```
ex:Conf1 co:possible [ a co:ConfigurationLink ;
    co:specToBeAdded r:Diesel : rdfs:label "Diesel";
    co:linkedConf ex:Conf1PlusDiesel .]
```

Proposing the selection of several specifications at once. Let us note that this model supports the selection of several specifications at once. This can be useful from a marketing point of view, as an emphasis on certain packs of specifications, or on certain full featured configurations:

```
ex:Conf3
    co:chosenSpec r:Model1;
    co:possible [ a co:ConfigurationLink ;
        rdfs:label "Over-equipped configuration!" ;
        co:specToBeAdded r:AirCond, r:RadioMP3, r:SunRoof ;
        co:linkedConf ex:overEquippedConf .].
```

co:alternative. A user may want to change one of its previous selections. This property lists those of the co:chosenSpec, which can be changed: it links the configuration to a similar one, with one of the co:chosenSpec removed or changed. This property may not be used when the chosen specification in question happens to be implied by the other choices. For instance, on ex:Conf1PlusDiesel, the r:Diesel can be replaced by r:Gasoline:

```
ex:Conf1PlusDiesel co:alternative [
    a co:ConfigurationLink ;
    co:specToBeRemoved r:Diesel ;
    co:specToBeAdded r:Gasoline ;
    co:linkedConf ex:Conf1PlusGasoline .].
```

co:impossible. When a specification is not compatible with a configuration, the configuration engine can nevertheless provide a way to select it - of course, at the cost of discarding some of the previous selections; there is a conflict, to be resolved by removing or changing some of the co:definingChoice(s).

```
ex:Conf1 co:impossible [
    a co:ConfigurationLink ;
    co:specToBeAdded r:SunRoof ;
    co:linkedConf ex:Conf1WithResolvedConflict .].
```

co:defaultSpec. Specification included by default in a Completely Defined Product matching this configuration.

co:lexicon. Used in particular to link a Configuration to the corresponding Lexicon. A way to get the variables definition, an information an application can use, for instance to make explicit the fact that specifications corresponding to a given variable are alternatives ones, e.g. to display a menu with radio buttons to choose the fuel type from.

Price. A configuration has a starting price, corresponding to the price of the cheapest product matching this configuration. We use the gr:hasPriceSpecification to state the starting price of a Configuration:

```
ex:Conf1 gr:hasPriceSpecification [
    a gr:UnitPriceSpecification ; gr:hasCurrency "EUR" ;
    gr:hasMinCurrencyValue "9000.00"^^xsd:float. ].
```

Starting prices of the linked configurations can be embedded within the RDF data returned when dereferencing the configuration:

```
ex:Conf1 co:possible [ a co:ConfigurationLink ;
    co:specToBeAdded r:Diesel ;
    co:linkedConf ex:Conf1PlusDiesel .].
ex:Conf1PlusDiesel gr:hasPriceSpecification [
    a gr:UnitPriceSpecification ; gr:hasCurrency "EUR" ;
    gr:hasMinCurrencyValue "10000.00"^^xsd:float. ].
```

Completion. Any configuration can be completed. The co:minPriceCompletedConf property links to a completely defined product matching this configuration, at the same price.

5.4 Integration with GoodRelations

A configuration mainly describes a Partially Defined Product. As such, in GoodRelations terms, a co:Configuration is a gr:ProductOrServiceModel:

> an intangible entity that specifies some characteristics of a group of similar, usually mass-produced products, in the sense of a prototype.

The suffix "Model" may seem misleading when used for a Configuration, as it suggests something such as "Ford T", and not "Ford T with Air Conditioning and MP3 connection plug" (itself not a completely defined product - you still can choose, well, the color: it is a "prototype of similar products").

On the other hand, a configuration has a price. It may be seen as a commercial offer, or the expression of a customer's wish list. It can therefore be considered as a gr:Offering as well. Giving, as we do, gr:hasPriceSpecification the start price of a co:Configuration makes it a de facto gr:Offering. Also, the range depends on the vendor, a typical characteristic of an offer; e.g. two PC vendors both sell, say, PC intel core i7 2500K, 4GB RAM: this is a configuration; however they propose different disk capacities.

So, a Configuration can be considered as both a gr:ProductOrServiceModel and a gr:Offering.

5.5 Vocabularies for Specifications

This ontology is generic: it does not depend on the variables and specifications used to define a product, and it allows a publisher to use its own terms as specifications. This is an important point, as the whole purpose of the configuration process is to come out with an order for a completely defined product, which implies its definition in the manufacturing company's terms. On the other hand, there are shared vocabularies on the web for products. No technical obstacle prevents us from adding triples using terms coming from such vocabularies to the description of a Configuration. Example using the Vehicle Sales Ontology:

```
ex:Conf5 a co:Configuration ;
    co:chosenSpec r:Model1, r:Gasoline ;
    vso:fuelType dbpedia:Gasoline.
```

We won't go further into this question.

5.6 Indexing Configurations

The configuration ontology gives us the means to precisely describe ranges of customizable products, making it easy to crawl them. In section 3.3, we saw how

an agent can implement a text based searching mechanism with small indexes, and with calls to the configuration service. What about search engines, then? We expect them to index our products as a matter of course.

The harsh reality, though, is that ranges are huge. We can proudly announce the availability of our 10^{20} descriptions of completely defined products on the web of data, and of even more partially defined ones, yet this is far more than what the most obstinate robot can cope with. So, we cannot but give thought to the fact that indexing will be partial.

Basically, configuration will be indexed by specifications. The semantics of the properties used to describe a Configuration should be carefully taken into account when deciding on which specifications indexing will be based. For instance, if the values of the "co:possible" property were used to index configurations, queries searching for products containing several specifications could return matches that actually do not include their conjunction: spec1 and spec2 can both be individually compatible with a given configuration, while spec1 and spec2 together is impossible. Or, they could get displayed at a lower price than the true one: the start price of a configuration generally increases when options are added. The only way to return accurate results would be to query the configuration service at runtime; while this is a simple thing for a specialized agent to do, search engines will not. As an other example, indexing configurations with chosen and implied specifications only would require to build a very large index, to get matches for searches involving many specifications. The best solution probably uses the union set of the values of co:chosenSpec, co:impliedSpec and co:defaultSpec.

Of course we do not know how search engines will proceed. We enable them to crawl the dataset, either starting just from its root (the "empty configuration"), or from any configuration, and following links whose semantics is precisely defined in the co:ConfigurationLink class. We provide them with enough information to customize their strategies. For instance, they can choose which links they follow. Not all specifications are of equal interest: the sun roof, the MP3 connector, etc. are probably more important - for a customer as well as for a search engine - than, say, the color of the ashtray.

On the other hand, the "sitemap" file of the web site is the place for the publisher to list configurations the indexing robots should consider first. A still unanswered question is: which configurations should be included in the sitemap file to get the most of it from a marketing point of view? Clearly, the choice should be driven by marketing data: for instance, which specifications and configurations should be "pushed" toward the customer?

6 Benefits

6.1 Improved Architecture

As noted in section 4, the access we historically provided to the functionalities of our configuration engine was through a java API. Switching to a REST based API brought its own benefits. Before this change, our configuration engine and

associated PRS data were duplicated in a number of different applications: web sites, salesman's assistant, etc. We improved our architecture by having it now really web based. Data and configuration engine are centralized on one server, accessed by all applications needing configuration functionalities. Updates are easier, server resources are shared, and the web architecture ensures scalability.

6.2 Reduced Development Costs of Client Applications

The development of several new client applications is on its way, and the costs are much lower than with our previous Java API: the GUI developer does not have to understand the concepts underlying configuration, nor (for the larger part) to learn an API. Basically, she just has to display the links found in the data.

6.3 Benefits of Universal Configuration Identifiers

Configurations truly deserve their status of first class objects. They represent Partially Defined Products. They also capture the exact expression of the customer's wish list, constrained by the definition of the range: a very important point of concern from a marketing point of view! Global identifiers for configurations may be put to a number of uses, most of which increase the visibility of the commercial offer. To name a few:

Tagging web content, defining links. Configurations may be used in describing web pages, can be the subject of clickable ads, etc.

Easily sharing configurations between applications, devices, media. Identifying configurations with URIs allows for their easy sharing between corporate systems: web site, salesman's assistant, ordering system, etc., as well as outside ones such as social networks. This results in improved fluidity of the e-business processes. As a proof of concept, we developed prototype software showing how a potential customer can begin a configuration by clicking on an ad or decoding a QR code in a billboard; modify it on her smartphone or PC; exchange it with members of her family; share it on FaceBook; have it transferred to the salesman's assistant when she finally goes to a shop, and have it converted to an order. By the way it emerged that integration with Facebook OpenGraph has applications of interest to marketing people.

6.4 e-business Use Case Example: Targeted ads

Agents knowledgeable about the buying habits and preferences of consumers can use this data to generate ads matching their possible wishes better. For instance, if a user, known to be young and accustomed to buying and downloading music, issues a query about cars, display an ad for a small car with an MP3 adaptor.

7 Conclusion

Data about customizable products can be published effectively as Linked Data. We described the corresponding service and ontology. Most, if not all, configurator applications on the web could be modified with relative ease, to publish data that way. It gets us accurate descriptions of complex ranges of products, which can be crawled and understood by simple agents: all reasoning takes place inside the service publishing the data, its complexity hidden from the clients. For search engines, the number of configurations is challenging - we added more than 10^{20} of them to the web of data - but the linked nature of the dataset should be sufficient to use it effectively. As for specialized agents, we expect them to offer new functionalities with configuration data, such as comparing ranges based on customer's interests; and in the end, to answer the question: what are the prices and CO2 emission levels of small cars with gasoline engines, sun-roofs, air conditioning, and MP3 adapters?

References

1. Hepp, M.: GoodRelations: An Ontology for Describing Products and Services Offers on the Web. In: Gangemi, A., Euzenat, J. (eds.) EKAW 2008. LNCS (LNAI), vol. 5268, pp. 329–346. Springer, Heidelberg (2008)
2. Badra, F., Servant, F.P., Passant, A.: A Semantic Web Representation of a Product Range Specification based on Constraint Satisfaction Problem in the Automotive Industry. In: OSEMA Workshop ESWC (2011), `http://ceur-ws.org/Vol-748/paper4.pdf`
3. Pargamin, B.: Vehicle Sales Configuration: the Cluster Tree Approach. In: ECAI Workshop on Configuration (2002)