Integrating Semantic Web Technologies and ASP for Product Configuration

Stefan Bischof[1] and Gottfried Schenner[1] and Simon Steyskal[1] and Richard Taupe[1,2]

Abstract. Currently there is no single dominating technology for building product configurator systems. While research often focuses on a single technology/paradigm, building an industrial-scale product configurator system will almost always require the combination of various technologies for different aspects of the system (knowledge representation, reasoning, solving, user interface, etc.) This paper demonstrates how to build such a hybrid system and how to integrate various technologies and leverage their respective strengths.

Due to the increasing popularity of the industrial knowledge graph we utilize Semantic Web technologies (RDF, OWL and the Shapes Constraint Language (SHACL)) for knowledge representation and integrate it with Answer Set Programming (ASP), a well-established solving paradigm for product configuration.

1 INTRODUCTION

The way large organizations manage their data is subject to trends. Currently the (industrial) knowledge graph is gaining popularity [13]. The term “knowledge graph” was coined by Google [20] in the context of search engines. This was adopted by the industry to describe a system that manages the data of a company with a graph-based formalism like RDF and combines it with reasoning (e.g. OWL) and machine learning [3]. Having all internal data of the company accessible in one format solves the problem of isolated data silos often found in large corporations and allows the combination of internal data with external data e.g. from the Linked Open Data cloud [4].

Knowledge graphs rely heavily on methods and technologies developed by the Semantic Web community. Given the current acceptance in the industry for these technologies it is a good time for the product configuration community to revisit these technologies. In this paper we demonstrate how to use Semantic Web technologies for product configuration and how to integrate them with established solving technologies for product configuration like Answer Set Programming.

Why is the use of open standards and technologies like the Semantic Web technology stack so important? In our experience using a configurator with a vendor-specific language for specifying product configuration problems is the equivalent of a data silo in data management. It is very hard to switch from one configurator vendor to another, if both systems use their own proprietary specification language. Also in the product configuration community there is currently no standard way to specify product configuration problems although the topic of ontologies and product configuration is over 20 years old (cf. [21]) and pre-dates the Semantic Web.

In Section 2 we describe the technologies used for this paper. In Section 3 we show how to define a product configurator knowledge base with RDF and SHACL, that allows checking of constraints and interactive solving.

For solving product configuration problems, RDF and SHACL are combined with Answer Set Programming in Section 4. In Section 5 we illustrate how reasoning with OWL can help to integrate the product configuration solutions into the knowledge graph and facilitate reuse of ontologies.

In Section 6 we give a brief overview of the systems used for the examples and in Section 7 we conclude.

2 PRELIMINARIES

The proposed approach builds heavily on Semantic Web standards and technologies. Instance data is represented as RDF triples, domain models are mapped to domain-dependent ontologies/vocabularies and queries are formulated in SPARQL [19].

2.1 RDF+SHACL

The Resource Description Framework (RDF) [6] is a both human-readable and machine-processable framework for describing and representing information about resources. In RDF every resource is identified by an IRI, and information about resources is represented in form of triples with $I \cup B \times I \times I \cup B \cup L$, where I, B, L denote IRIs, blank nodes (nodes that do not have a corresponding IRI and which are mainly used to describe special types of resources without explicitly naming them) and RDF literals (e.g., strings, integers, etc.) respectively.

The Shapes Constraint Language (SHACL) [9] – a W3C Recommendation since 2017 – is a language for validating RDF graphs against a set of constraints. Its validation process is built around the notion of *Data Graphs* (RDF graphs that contain the data that has to be validated), and *Shapes Graphs* (RDF graphs containing shape definitions and other information that is used to perform the validation of the *Data Graphs*).

[1] Siemens AG Österreich, Corporate Technology, Vienna, Austria bischof.stefan@siemens.com, gottfried.schenner@siemens.com, simon.steyskal@siemens.com, richard.taupe@siemens.com
Author names are given in alphabetical order.
[2] Alpen-Adria-Universität, Klagenfurt, Austria

2.2 Ontologies

The *OWL 2 Web Ontology Language* (OWL) [12] is a declarative knowledge representation formalism standardized by the W3C in 2012. OWL is a language to represent *ontologies* and is based on description logics.

An ontology describes things (*individuals*), sets of individuals (*classes*), relations between individuals (*object properties*) and attributes of individuals (*data properties*). Based on these, OWL provides class and property constructors to define complex classes and properties, e.g., intersection or union of classes. Eventually, with OWL *axioms*, we can define how classes (and properties) are related to each other, e.g., subclasses, equivalent classes, or disjoint classes.

OWL ontologies can be serialized in one of several syntaxes. In this paper we use Turtle [2] syntax for serializing both OWL and SHACL.

2.3 Validation vs. Inference

In the Semantic Web, the tasks of (*i*) constraint validation and (*ii*) reasoning are grounded on different semantics. While the latter usually operates under the *Open World Assumption (OWA)* (i.e., a statement cannot be assumed to be false if it cannot be proven to be true [16]) and the *Non-Unique Name Assumption (nUNA)* (i.e., the same resource can have multiple names), validation adheres to the *Closed World Assumption (CWA)* (i.e., a statement is assumed to be false if it cannot be proven to be true) and requires that different names identify different objects (i.e., it makes the *Unique Name Assumption (UNA)*).

Those differences are illustrated in Figure 1 where OWL is used for specifying a cardinality constraint for individuals of type `:ElementA` allowing them to refer to exactly one module of type `:ModuleI` only.

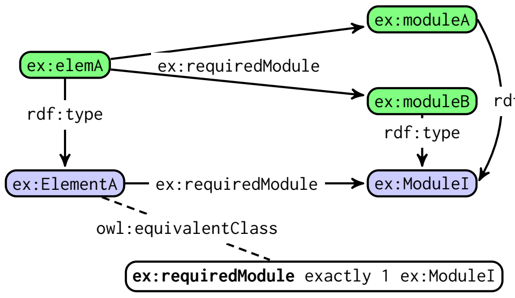

Figure 1. `ex:elemA` violates its cardinality constraint only under UNA/CWA, but not under nUNA/OWA where it could be inferred that `ex:moduleA` and `ex:moduleB` represent the same resource, thus not violating the cardinality constraint.

Class `ex:ElementA` is defined to be equal to an anonymous class having exactly one `ex:ModuleI` associated via `ex:requiredModule`, hence every individual of type `ex:ElementA` is only allowed to refer to one specific module. Even though `ex:elemA` is violating its cardinality constraint (because it is associated to more than one module via `ex:requiredModule`) an OWL reasoner would not be able to detect a violation, but instead infers that `ex:moduleA` and `ex:moduleB` are representing the same thing.

2.4 ASP

Answer Set Programming (ASP) is a declarative programming paradigm. Instead of specifying how to find a solution to a problem in terms of an imperative algorithm, in ASP the problem itself is specified in the form of a logic program. We restrict our introduction to ASP to core concepts needed to understand this paper and refer to [5,10,11] for more details.

A program P is a finite set of rules of the form

$$\mathtt{h} \;\mathtt{:-}\; \mathtt{b}_1,\; \ldots,\; \mathtt{b}_m,\; \mathbf{not}\; \mathtt{b}_{m+1},\; \ldots,\; \mathbf{not}\; \mathtt{b}_n.$$

where $\mathtt{h}$ and $\mathtt{b}_1, \ldots, \mathtt{b}_m$ are positive literals (i.e. atoms) and $\mathbf{not}\ \mathtt{b}_{m+1}, \ldots, \mathbf{not}\ \mathtt{b}_n$ are negative literals. An atom is either a classical atom or a cardinality atom. A classical atom is an expression $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ where $\mathtt{p}$ is an n-ary predicate and $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are terms. A term is either a variable (whose name starts with an upper-case character or an underscore) or a constant (which can be a number or a string). A literal is either an atom α or its default negation $\mathbf{not}\ \alpha$. Default negation refers to the absence of information, i.e. an atom is assumed to be false as long as it is not proven to be true. Thus, ASP makes the Closed World Assumption.

A *cardinality atom* is of the form

$$\mathtt{l} \prec_{\mathsf{l}} \{\; \mathtt{a}_1 : \mathtt{l}_{1_1}, \ldots, \mathtt{l}_{1_m};\; \ldots;\; \mathtt{a}_n : \mathtt{l}_{n_1}, \ldots, \mathtt{l}_{n_o} \} \prec_{\mathsf{u}} \mathtt{u}$$

where each structure $\mathtt{a}_i : \mathtt{l}_{i_1}, \ldots, \mathtt{l}_{i_m}$ is a *conditional literal* in which $\mathtt{a}_i$ (the head of the conditional literal) and all $\mathtt{l}_{i_j}$ are classical literals, and $\mathtt{l}$ and $\mathtt{u}$ are terms representing non-negative integers indicating lower and upper bound. If one or both of the bounds are not given, their defaults are used, which are 0 for $\mathtt{l}$ (and $\leq$ for $\prec_{\mathsf{l}}$) and ∞ for $\mathtt{u}$ (and $\leq$ for $\prec_{\mathsf{u}}$).

$H(r) = \{\mathtt{h}\}$ is called the *head* of the rule, and $B(r) = \{\mathtt{b}_1, \ldots, \mathtt{b}_m, \mathbf{not}\ \mathtt{b}_{m+1}, \ldots, \mathbf{not}\ \mathtt{b}_n\}$ is called the body of the rule. A rule with empty head is a *constraint* and is used to filter out invalid solutions. A rule with empty body is called *fact*, its head holds unconditionally in a satisfiable program.

There are several ways to define the semantics of an answer-set program, i.e. to define the set of answer sets $AS(P)$ of an answer-set program P. An overview is provided by [11]. Informally, an answer set A of a program P is a subset-minimal model of P (i.e. a set of atoms interpreted as *true*) which satisfies the following conditions: All rules in P are satisfied by A; and all atoms in A are "derivable" by rules in P. A rule is satisfied if its head is satisfied or its body is not. A cardinality atom is satisfied if $l \prec_{\mathsf{l}} |C| \prec_{\mathsf{u}} u$ holds, where C is the set of head atoms in the cardinality literal whose conditions (e.g. $\mathtt{l}_{i_1}, \ldots, \mathtt{l}_{i_m}$ for $\mathtt{a}_i$) are satisfied and which are satisfied themselves.

Most ASP systems split the solving process into grounding and solving. The former part produces the grounding of a program, i.e. its variable-free equivalent. Thereby, the variables in each rule of the program are substituted by constants. The latter part then solves this propositional encoding.

To briefly illustrate ASP by means of a small example, consider the following program:

```
triple("ex:ElementA", "rdfs:subClassOf", "ex:Element").
rdfs_subClassOf(Sub,Sup) :-
 triple(Sub, "rdfs:subClassOf", Sup).
class(Sub) :- rdfs_subClassOf(Sub,Sup).
class(Sup) :- rdfs_subClassOf(Sub,Sup).
n_element_types(N) :-
 N = { rdfs_subClassOf(Sub,"ex:Element") : class(Sub) }.
```

This program contains one fact representing an RDF triple[3] and two rules that derive new atoms from it. The single answer set of this program is:

```
{
 triple("ex:ElementA","rdfs:subClassOf","ex:Element"),
 rdfs_subClassOf("ex:ElementA","ex:Element"),
 class("ex:ElementA"), class("ex:Element"),
 n_element_types(1)
}
```

3 PRODUCT CONFIGURATION WITH SHACL

The following describes the product configuration terminology from [8] adapted to RDF + SHACL.

Definition 1 *(Configuration Model) The (SHACL) configuration model CONFMODEL = (SHAPEGRAPH, RDF) consists of a SHACL shapes graph and an ontology/RDFS schema defining all the used classes and properties in the configuration model.*

Definition 2 *(User requirements) The (SHACL) user requirements USERREQ = (SHACLCONSTRAINTS, SUBGRAPH) consists of additional SHACL constraints and an initial RDF subgraph.*

Definition 3 *(Configuration Task) The configuration task CONFIGTASK = (CONFMODEL, USERREQ) represents the input of one concrete configuration problem.*

Definition 4 *(Configuration) A configuration (solution) of a configuration task CONFIGTASK is an RDF graph, which satisfies the SHACL constraints of CONFMODEL and USERREQ and contains SUBGRAPH of USERREQ as a sub graph.*

The configuration model defines the constraints that all configurations must satisfy. The ontology is used to identify the classes and properties relevant for the current configuration task. In a large knowledge graph (RDF store/graph database) there can be thousands of concepts and properties and typically only a small subset is relevant for product configuration. Even within the classes relevant for product configuration not all will be relevant in the current configuration task as the configuration task of large artefacts is typically split into smaller configuration tasks like hardware configuration and software configuration. Although the tasks are separated, the resulting configurations will reside in the same knowledge graph and will be linked, e.g., a hardware component will be related to the required software driver.

In the following we will illustrate the product configuration concepts by means of an example.

3.1 Running example

For ease of comparison we use the example of [18] as a running example. This is an abstract example of a typical hardware configuration problem found in industry with some key features like a component taxonomy, cardinality restrictions for relations between component types, etc.

[3] For conciseness, we use RDF prefixes also when we present ASP encodings in this paper, even though the real encodings contain full IRIs. For example, the string `"rdfs:subClassOf"` stands for `"http://www.w3.org/2000/01/rdf-schema#subClassOf"`.

In our example domain there may be different types of elements, which are controlled by hardware modules. Each hardware module must be in a frame and a frame must be mounted on a rack. More specifically, the constraints of the domain are:

- There are four disjoint types of elements (ElementA-ElementD).
- There are five disjoint types of modules (ModuleI-ModuleV).
- There are two disjoint types of racks (RackSingle, RackDouble).
- An ElementA/B/C/D requires exactly one/two/three/four ModuleI/II/III/IV respectively, i.e., an ElementB must have only and exactly two ModuleIIs associated via the requiredModule property.
- A ModuleV cannot have an element, all other modules must be required by an element (via the requiredModule property).
- A RackSingle must contain exactly four frames, a RackDouble must contain exactly eight frames.
- A frame must be mounted on a rack.
- A frame can contain up to six modules.
- A module must be mounted on a frame.
- Whenever a frame contains a module of type ModuleII, it must also contain one of type ModuleV.

The ontology of our running example is illustrated in Fig. 2.

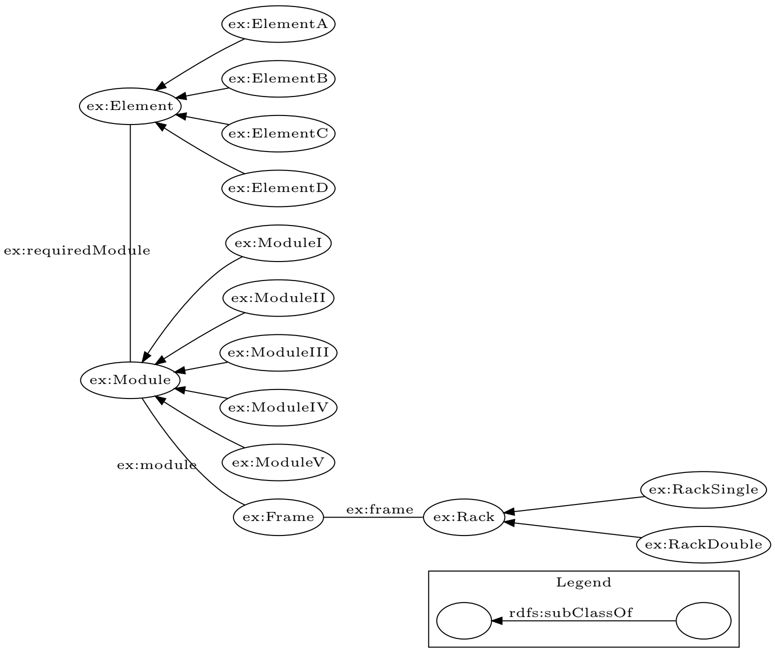

Figure 2. Racks ontology RDF graph

The following shows an excerpt of the ontology in turtle format:

```
ex:ElementA rdfs:subClassOf ex:Element .
ex:ElementB rdfs:subClassOf ex:Element .
ex:ElementC rdfs:subClassOf ex:Element .
ex:ElementD rdfs:subClassOf ex:Element .
```

```
ex:ModuleI rdfs:subClassOf ex:Module .
ex:ModuleII rdfs:subClassOf ex:Module .
ex:ModuleIII rdfs:subClassOf ex:Module .
ex:ModuleIV rdfs:subClassOf ex:Module .
ex:ModuleV rdfs:subClassOf ex:Module .

# defines the objectproperty relating elements
# to their required modules
# the cardinality constraints for the objectproperty
# are defined with SHACL constraints
:requiredModule rdf:type owl:ObjectProperty ;
  rdfs:domain :Element ;
  rdfs:range :Module .
```

3.2 SHACL constraints

In the subsequent paragraphs, we illustrate some types of constraints supported by SHACL that are relevant in the configuration domain and refer to [9] for a full account of SHACL constraints.

Cardinality constraints A typical class of constraints for industrial product configuration problems are cardinality constraints. Whereas in customer product configuration problems simple yes/no or mandatory/optional constraints between components/features are often sufficient, complex cardinality constraints are common in industrial configuration problems, e.g., in our example each element class requires a different number and type of attached module. In SHACL these restrictions can be expressed with qualified cardinality constraints. For example the constraint that each instance of `ex:ElementB` requires exactly 2 `ex:ModuleII` can be expressed like this:

```
ex:ElementBRequiredModuleShape
  a sh:NodeShape ;
  sh:targetClass ex:ElementB ;
  sh:property [
    sh:path ex:requiredModule ;
    sh:minCount 2 ;
    sh:maxCount 2 ;
    sh:class ex:ModuleII ;
  ] .
```

Cardinality constraints are a powerful mechanism of product configuration models. With them it is possible to express mandatory/optional, requires, and part/subpart relationships. Almost all the constraints of our running example can be expressed with cardinality constraints.

Completeness of Taxonomy Typically in a configuration model it is required that every object of a configuration must be an instance of a leaf class in the taxonomy, e.g., if a configuration contains a rack, it must be known whether it is a RackSingle or a RackDouble. In SHACL this constraint can be expressed as:

```
# subclass inheritance disjoint and complete
ex:RackSubclassShape
  a sh:NodeShape ;
  sh:targetClass ex:Rack ;
  sh:message
        "A Rack must be either of
        type ex:RackSingle or ex:RackDouble" ;
  sh:xone (
     [  sh:class ex:RackSingle ]
     [  sh:class ex:RackDouble ]
  ) .
```

The reason for requiring completeness is that a configuration should be a complete description of all components the configured artefact contains. If the exact type of a component is not known, the solution may be underspecified.

There are some scenarios like ETO (engineer to order) where this restriction does not apply. In these cases some parts of the configured artefact are deliberately left unspecified because it is the task of the engineering department to come up with a solution, e.g., in our example we could add as a SHACL constraint that every rack needs a power supply but let the exact type of power supply unspecified. It is then the task of the engineering department to find a suitable power supply for the given configuration.

3.3 Checking constraints

Once we have defined the SHACL configuration model we can use it to check if a given RDF graph is a valid configuration. For example checking the RDF graph consisting of the single triple `ex:EB a ex:ElementB .` against previously introduced SHACL constraints produces the following validation result:

```
[
    a sh:ValidationResult ;
    sh:resultSeverity sh:Violation ;
    sh:sourceConstraintComponent
           sh:MinCountConstraintComponent ;
    sh:sourceShape _:n236 ;
    sh:focusNode ex:EB ;
    sh:resultPath ex:requiredModule ;
    sh:resultMessage "Less than 2 values" ;
] .
```

3.4 Interactive solving

In an interactive setting the user of the configurator will start with a non-empty subgraph and the system will indicate all currently violated SHACL constraints. The following shows a (fictional) session between user and configurator (conf), where the user asserts additional triples and the configurator reports constraint violations:

```
user: ex:E1 a ex:Element .
conf: ex:E1 must be oneof {ElementA, ..., ElementD}
user: ex:E1 a ex:ElementA .
conf: ex:E1 requires one ModuleI
user: ex:E1 ex:requiredModule ex:M1 .
user: ex:M1 a ex:ModuleI .
conf: ex:M1 requires a Frame
user: ex:F1 ex:module ex:M1 .
user: ex:F1 a ex:Frame .
conf: ex:F1 requires a Rack
user: ex:R1 ex:frame ex:F1 .
user: ex:R1 a ex:RackSingle .
conf: ex:R1 requires 4 frames
user: ex:R1 ex:frame ex:F2; ex:F3; ex:F4 .
user: ex:F2 a ex:Frame .
user: ex:F3 a ex:Frame .
user: ex:F4 a ex:Frame .
conf: No constraints violated
```

The resulting configuration of this configuration task is shown in Fig. 3 .

Although that kind of interaction might be sufficient for a domain expert, it is clear that the interactive configuration task shown above can be improved. There are only two points in the configuration task where a decision by the user is required. The first decision is to create an ElementA and the second decision is to choose between an RackSingle and a RackDouble. All other statements could be automatically derived by a reasoner.

Some simple statements can be derived with an RDFS reasoner. For example, `ex:F2 a ex:Frame .` follows from the

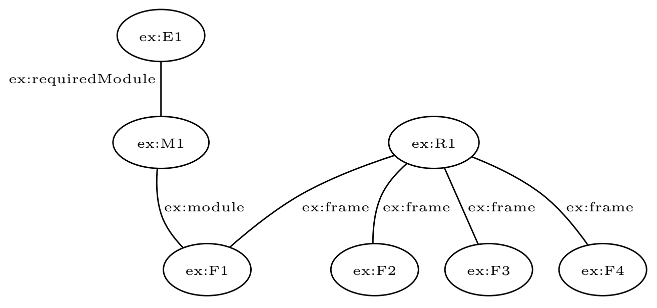

Figure 3. Racks configuration example

fact that `ex:F2` is an object in an `ex:frame` property and this property is defined to have the range `ex:Frame`.

Other SHACL constraints cannot be repaired by a typical Semantic Web reasoner because they require the creation of new entities. For these cases SHACL rules can be applied. The following example illustrates how SHACL rules could be utilized to "repair" a SHACL constraint violation. This rule creates frames for a rack until the required number of frames was created.

```
ex:RackRuleShape
  a sh:NodeShape ;
  sh:targetClass ex:RackSingle ;
  sh:rule [
    a sh:SPARQLRule ;
    sh:prefixes ex: ;
    sh:construct """
      PREFIX ex: <http://example.org/confws2018#>
      CONSTRUCT {
        this ex:frame _:new .
        _:new a ex:Frame .
      }
      WHERE { }
    """ ;
    sh:condition [ sh:not ex:RackSingleFrameShape ] ;
  ] .
```

4 SOLVING SHACL CONFIGURATIONS WITH ASP

Unfortunately, SHACL rules are a relatively new concept and there are no rule engines that support backtracking. Therefore one must revert to more established solving techniques for product configuration. For this paper we decided to use ASP.

As a basic proof of concept, we demonstrate the generic translation of SHACL constraints to ASP for concepts needed in the example illustrated in Section 3.1. As such, this demonstration includes only a subset of the SHACL Core Validators [9, Appendix D]. Most of the other validators can be expressed in ASP as well[4].

4.1 General Concepts

We represent RDF triples in ASP as instances of the ternary predicate `triple`, whose arguments are `Subject`, `Predicate`, and `Object`. For convenience, triples of certain RDF predicates are mapped to smaller ASP atoms by a set of projection rules:

```
rdfs_subClassOf(Sub,Sup) :-
 triple(Sub,"rdfs:subClassOf",Sup).
a(Instance,Class) :-
 triple(Instance,"rdf:type",Class).
sh_target_class(S, C) :- triple(S, "sh:targetClass", C).
sh_node_property(NS, PS) :- triple(NS, "sh:property", PS).
sh_property_shape(PS) :- triple(NS, "sh:property", PS).
sh_property_shape(PS) :-
 triple(PS, "rdf:type", "sh:PropertyShape").
sh_property_minCount(PS, Min) :-
 triple(PS, "sh:minCount", Min).
sh_property_maxCount(PS, Max) :-
 triple(PS, "sh:maxCount", Max).
sh_xone(S, List) :-
 triple(S, "sh:xone", List).
```

To enable the reasoning component to not only rule out invalid solutions but also explain inconsistencies, we use `shacl_constraint_violated` atoms in the head of constraints together with SHACL messages to encode explanations in the form of `shacl_constraint_violation_message` atoms in answer sets:

```
sh_message(S,Msg) :-
  triple(S,"sh:message",Msg).
shacl_constraint_violation_message(I,S,Msg) :-
  shacl_constraint_violated(I,S), sh_message(S,Msg).
```

4.2 Property Shapes

Property shapes specify constraints that need to be fulfilled by nodes that are reached on a SHACL property path, which can be defined in various ways [9, Section 2.3]. These constraints and paths need to be mapped to ASP concepts to enable mapping of ASP constraints to the intended targets of a SHACL property. A subset of these paths is handled by the following encoding:

```
sh_property_path(PS, P) :-
 triple(PS, "sh:path", P), not sh_path_is_inverse(P).
sh_path_is_inverse(P) :- triple(P, "sh:inversePath", _).
sh_property_path_inv(PS, InvP) :-
 triple(PS, "sh:path", P),
 triple(P, "sh:inversePath", InvP).

shacl_property_target(PS, Trgt) :-
 sh_property_target_inst(PS, Inst, Trgt).
sh_property_target_inst(PS, Inst, Trgt) :-
 sh_property_shape(PS), sh_node_property(NS, PS),
 sh_property_path(PS, P), sh_targetClass(NS, Class),
 a(Inst, Class), triple(Inst, P, Trgt).
sh_property_target_inst(PS, Inst, Trgt) :-
 sh_property_shape(PS), sh_node_property(NS, PS),
 sh_property_path_inv(PS, InvP), sh_targetClass(NS, Class),
 a(Inst, Class), triple(Trgt, InvP, Inst).
```

4.3 Class Membership

As an example for a simple SHACL constraint, we provide an encoding for the condition that a value node must be a SHACL instance of a given type (cf. [9, Section 4.1.1]):

```
shape_constraint(S, class, C) :-
 triple(S, "sh:class", C).
shape_constraint_satisfied_inst(I, S, class, C) :-
 shape_constraint(S, class, C), a(I, C).
```

4.4 Cardinality Constraints

Based on the encoding fragments presented so far we are now able to encode detection of cardinality constraint violations (cf. [9, Section 4.2]):

```
shacl_constraint_violated(I,PS) :-
 sh_property_shape(PS), sh_node_property(NS,PS),
 sh_target_class(NS,C), a(I,C), sh_property_minCount(PS,Min),
 not Min { sh_property_target_inst(PS,I,T) }.
shacl_constraint_violated(I,PS) :-
 sh_property_shape(PS), sh_node_property(NS,PS),
 sh_target_class(NS,C), a(I,C), sh_property_maxCount(PS,Max),
 not { sh_property_target_inst(PS,I,T) } Max.
```

[4] Validators that rely on string operations or data types are difficult to encode in ASP.

4.5 Logic Constraints

Logic constraints like xone (cf. [9, Section 4.6]) can also be mapped to cardinality constraints:

```
sh_xone_inst(Inst, List) :- sh_xone(S, List),
 sh_target_class(S, Class), a(Inst, Class).
shacl_constraint_violated(Inst, S) :-
 sh_xone(S, List), sh_xone_inst(Inst, List),
 not 1 {
  shape_constraint_satisfied_inst(
   Inst, List, Constraint, Value
  ) : shape_constraint(List, Constraint, Value)
 } 1.
```

4.6 Solving

Given a translation of the SHACL constraints into ASP we can check if the given RDF graph is a valid configuration, i.e., ASP acts as an implementation of the SHACL validator. To enable solving, an additional generative program is needed. This generative program must be capable of enumerating all possible solutions within a certain scope. The generative program together with the translated SHACL constraints enables us to find a valid configuration, if one exists within the given scope. The following listing shows a generative program capable of enumerating all configurations consisting of racks and frames. The scope is controlled by blank nodes (e.g., `_:b1`). Every blank node can become a new component within the configuration.

```
bnode("_:b1").
bnode("_:b2").
bnode("_:b3").
bnode("_:b4").
bnode("_:b5").
bnode("_:b6").

configobj(O) :-
    triple(O,"a",C),
    configclass(C).

0 { triple(BNODE, "a", "ex:RackSingle") } 1 :-
    bnode(BNODE).
0 { triple(BNODE, "a", "ex:RackDouble") } 1 :-
    bnode(BNODE).
0 { triple(BNODE, "a", "ex:Frame") } 1 :-
    bnode(BNODE).

0 { triple(O1, "ex:frame", O2)} 1 :-
    configobject(O1),
    configobject(O2).

% answer set found
triple("_:b1","a","ex:Rack").
triple("_:b1","a","ex:RackSingle").
triple("_:b2","a","ex:Frame").
triple("_:b3","a","ex:Frame").
triple("_:b4","a","ex:Frame").
triple("_:b5","a","ex:Frame").
triple("_:b1","ex:frame","ex:b2").
triple("_:b1","ex:frame","ex:b3").
triple("_:b1","ex:frame","ex:b4").
triple("_:b1","ex:frame","ex:b5").
```

After an answer set has been found due to the triple notation it can be directly translated back to an RDF graph. This RDF graph is a solution of the configuration task and satisfies all SHACL constraints.

5 COMBINING SHACL/RDF WITH OWL REASONING

So far we have concentrated on closed world reasoning and the unique name assumption for checking and finding configurations. In practice there are applications for CWA/OWA and UNA/nUNA. As a use case for open world reasoning we will demonstrate how to reuse an existing ontology for our running example.

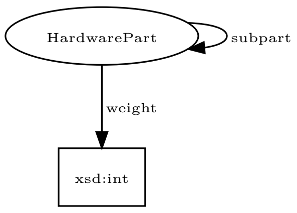

Figure 4. Hardware part Ontology (:hw)

5.1 Semantic differences of SHACL and OWL reasoning

Consider the constraint that a single rack has four frames. Due to closed world reasoning the SHACL constraint is violated in the example below. For a OWL reasoner the example is consistent, because OWL adheres to the open world assumption and thus an OWL reasoner would not be able to decide if there is another frame.

```
ex:R1 a ex:RackSingle .
ex:R1 ex:frame ex:F1 .
ex:R1 ex:frame ex:F2 .
ex:R1 ex:frame ex:F3 .
```

In the next example again a SHACL constraint will be violated, because there are 5 frames associated with a Rack and under the Unique Name Assumption they are all considered different entities. In OWL this example is consistent, because the unique name assumption is not applied and there is no statement indicating that, e.g., `ex:F1` and `ex:F2` are different entities.

```
ex:R1 a ex:RackSingle .
ex:R1 ex:frame ex:F1 .
ex:R1 ex:frame ex:F2 .
ex:R1 ex:frame ex:F3 .
ex:R1 ex:frame ex:F4 .
ex:R1 ex:frame other:FA .
```

5.2 Reusing ontologies with OWL reasoning

Modeling subpart relations can be surprisingly complicated (cf. [7]), but is necessary in many knowledge representation systems. For presentational reasons we opt for a simplistic ontology: there exist hardware parts which can have subparts which are also hardware parts. The *subpart* relation is intentionally not defined as transitive and thus expresses only direct subparts. Additionally each hardware part must have a weight (see Fig. 4). Having hardware parts modeled along this ontology, allows us to (recursively) compute the *total weight* of a hardware part by adding the weight of the part itself with the sum of the *total weights* of all (direct) subparts. However, recursion is not needed, and so we will instead compute the total weight of a hardware part by adding its own weight to the weight to all (transitively reachable) subparts. The following SPARQL query performs this computation for all hardware parts (naturally the variable `?part` in the `WHERE` clause could be replaced by the URI of some hardware part):

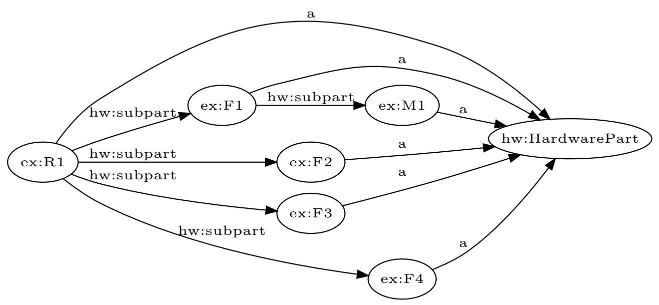

Figure 5. Configuration example mapped to hardware part ontology

```
SELECT ?part (SUM(?weight) as ?totalweight)
WHERE {
  ?part p:subpart*/p:weight ?weight .
} GROUP BY ?part
```

But before we can do that, we have to map the configuration created previously to this Hardware Part ontology. The mapping can be expressed by SHACL rules or a *mapping ontology*. Taking the latter approach we can hold off on the decision for a concrete implementation strategy. In our example we only have to define the object properties `ex:frame` and `ex:module` as subproperties of `p:subpart` in the mapping ontology. Taking all the ontologies (example, subpart, and mapping) and the racks configuration data into account, an OWL reasoner would (also) entail the subpart relationships depicted in Fig. 5. Depending on the OWL reasoner implementation these entailments are materialised or only inferred for relevant queries. Assuming the weights of the hardware parts themselves are also available (for example through a similar mapping to a product catalogue) we now have all the necessary parts to compute the total weight of all the hardware parts with the above SPARQL query.

Alternatively we could use a similar query to materialise the total weights in the RDF graph:

```
INSERT { ?part p:totalweight ?totalweight }
WHERE { {
  SELECT ?part (SUM(?weight) as ?totalweight)
  WHERE {
    ?part p:subpart*/p:weight ?weight .
  } GROUP BY ?part
} }
```

Using this mapping approach we can transparently map our instance data with relevant data from other (legacy) information systems via a knowledge graph and thus create a more complete knowledge base. We therefore are able to reuse ontologies like the Hardware Part ontology and the associated SPARQL queries in a modular manner.

6 EVALUATION

All the example RDF files, ontologies, ASP and Java programs are available in our open-source git repository[5]. For manipulating RDF the Apache Jena library version 3.5.0[6] was used. For checking SHACL constraints we relied on the TopBraid SHACL API version 1.1.0[7]. The SHACL examples can also be checked online interactively by using the SHACL playground[8]. The SHACL rule example has been implemented in Java. The translation of the SHACL example to ASP is also implemented in Java. For running the ASP programs we used the ASP solver clingo version 5.2.0[9]. The OWL ontologies were edited with the Protégé ontology editor[10] version 5.2. The classification example of 5 was verified with the integrated HermiT reasoner. The example was also executed with StarDog version 5.3.0[11] by using a SPARQL query with activated OWL reasoning. StarDog is a knowledge graph platform for the enterprise.

7 CONCLUSION

We have used Semantic Web technologies in the past, but only for specific purposes like data integration [17]. Product configuration knowledge bases require closed world reasoning and the default open world reasoning of OWL made it cumbersome to be used for that task in our experience. We found that SHACL closes this gap and have demonstrated in this paper how to define a configurator knowledge base just with RDF+SHACL. Together with constraint validation such a system can be the basis of a simple interactive configurator. SHACL rules can enhance the user experience by deriving additional knowledge. We do not expect SHACL to be *the* language for specifying product configuration problems in combination with ontologies but it is a step into the right direction.

Because of the lack of backtracking SHACL rule engines, SHACL rules can currently not be used to solve configurations except for trivial examples. A solver for product configuration problems finds a model for the product configuration model. This is not a reasoning task supported by a typical Semantic Web reasoner. The main task for a Semantic Web reasoner is classification and determining consistency[12].

Therefore we resorted to ASP for solving the configuration problem. To demonstrate the feasibility of the approach we translated the subset of SHACL required to solve our simple example domain. We will continue to evaluate our approach on more sophisticated domains e.g. by adding arithmetic constraints. Because the semantic of SHACL can be defined by SPARQL and the expressive power of SPARQL [1] is sufficient for typical product configuration domains we expect no conceptual difficulties.

The main challenge will be the automatic translation of the SHACL constraints into ASP. In the future we want to adopt a more generic approach by building on the work already done for SPARQL and ASP [14, 15]. Of course such a translational approach is not restricted to ASP. The same could be applied to SAT, CSP or any other solving paradigm for product configuration.

An important topic for the future will be how to identify the relevant information for product configuration in the knowledge graph. This includes how to (semi-)automatically identify the parts that are currently available for configuring a

[5] https://github.com/siemens/ProductConfigurationWithSHACL
[6] https://jena.apache.org/
[7] https://github.com/TopQuadrant/shacl

[8] http://shacl.org/playground/
[9] https://potassco.org/
[10] https://protege.stanford.edu/
[11] https://www.stardog.com/
[12] A Semantic Web reasoner might eventually produce a model internally for proving consistency, but this is normally not available to the caller of the reasoner.

product, how to relate the product configuration ontologies to other relevant ontologies for product line management, enterprise resource planning etc.

ACKNOWLEDGEMENTS

This work was partially conducted within the scope of DynaCon (FFG-PNr.: 861263), which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program “ICT of the Future” between November 2017 and April 2020. More information: https://iktderzukunft.at/en/

REFERENCES

[1] Renzo Angles and Claudio Gutierrez, ‘The expressive power of sparql’, in *The Semantic Web - ISWC 2008*, eds., Amit Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy Finin, and Krishnaprasad Thirunarayan, pp. 114–129, Berlin, Heidelberg, (2008). Springer Berlin Heidelberg.

[2] David Beckett, Tim Berners-Lee, Eric Prud’hommeaux, and Gavin Carothers, ‘RDF 1.1 Turtle – terse RDF triple language’, Recommendation, W3C, (February 2014).

[3] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger, ‘Swift logic for big data and knowledge graphs’, in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, ed., Carles Sierra, pp. 2–10. ijcai.org, (2017).

[4] Christian Bizer, Tom Heath, and Tim Berners-Lee, ‘Linked data - the story so far’, *Int. J. Semantic Web Inf. Syst.*, **5**(3), 1–22, (2009).

[5] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński, ‘Answer set programming at a glance’, *Communications of the ACM*, **54**(12), 92–103, (2011).

[6] Richard Cyganiak, David Wood, and Markus Lanthaler, ‘Resource Description Framework (RDF)’, Recommendation, W3C, (February 2014). Available at https://www.w3.org/TR/rdf11-concepts/.

[7] Mariano Fernández-López, Asunción Gómez-Pérez, and Mari Carmen Suárez-Figueroa, ‘Selecting and customizing a mereology ontology for its reuse in a pharmaceutical product ontology’, in *Proceedings of the 2008 Conference on Formal Ontology in Information Systems: Proceedings of the Fifth International Conference (FOIS 2008)*, pp. 181–194, Amsterdam, The Netherlands, The Netherlands, (2008). IOS Press.

[8] Lothar Hotz, Alexander Felfernig, Markus Stumptner, Anna Ryabokon, Claire Bagley, and Katharina Wolter, ‘Chapter 6 - configuration knowledge representation and reasoning’, in *Knowledge-Based Configuration*, eds., Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen, 41–72, Morgan Kaufmann, Boston, (2014).

[9] Holger Knublauch and Dimitris Kontokostas, ‘Shapes Constraint Language (SHACL)’, Recommendation, W3C, (July 2017). Available at https://www.w3.org/TR/shacl/.

[10] Vladimir Lifschitz, ‘What Is Answer Set Programming?’, in *Twenty-Third AAAI Conference on Artificial Intelligence*, (2008).

[11] Vladimir Lifschitz, ‘Thirteen Definitions of a Stable Model’, in *Fields of Logic and Computation*, eds., Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, volume 6300 of *Lecture Notes in Computer Science*, 488–503, Springer, Berlin, Heidelberg, (2010).

[12] W3C OWL Working Group, ‘Owl 2 web ontology language: Document overview’, Recommendation, W3C, (October 2009). Available at http://www.w3.org/TR/owl2-overview/.

[13] *Exploiting Linked Data and Knowledge Graphs in Large Organisations*, eds., Jeff Z. Pan, Guido Vetere, José Manuél Gómez-Pérez, and Honghan Wu, Springer, 2017.

[14] Axel Polleres, ‘From SPARQL to rules (and back)’, in *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, eds., Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, pp. 787–796. ACM, (2007).

[15] Axel Polleres and Johannes Peter Wallner, ‘On the relation between sparql1. 1 and answer set programming’, *Journal of Applied Non-Classical Logics*, **23**(1-2), 159–212, (2013).

[16] Raymond Reiter, ‘On closed world data bases’, in *Logic and Data Bases*, eds., Hervé Gallaire and Jack Minker, 55–76, Springer, (1978).

[17] Gottfried Schenner, Stefan Bischof, Axel Polleres, and Simon Steyskal, ‘Integrating distributed configurations with RDFS and SPARQL’, in *Configuration Workshop*, volume 1220, pp. 9–15, (2014).

[18] Gottfried Schenner and Richard Taupe, ‘Techniques for solving large-scale product configuration problems with ASP’, in *Proceedings of the 19th International Configuration Workshop*, eds., Linda L. Zhang and Albert Haag, pp. 12–19, La Défense, France, (2017).

[19] Andy Seaborne and Steven Harris, ‘SPARQL 1.1 Query Language’, Recommendation, W3C, (March 2013). Available at https://www.w3.org/TR/sparql11-query/.

[20] Amit Singhal, ‘Introducing the knowledge graph: things, not strings’, *Official Google Blog*, (2012). Available at https://www.blog.google/products/search/introducing-knowledge-graph-things-not.

[21] Timo Soininen, Juha Tiihonen, Tomi Männistö, and Reijo Sulonen, ‘Towards a general ontology of configuration’, *AI EDAM*, **12**(4), 357–372, (1998).