Experience-based Quality Assessment of Distributed Knowledge Graphs

Joachim Baumeister[1]

Abstract: This paper introduces an experience-based approach for the evaluation of distributed knowledge graphs. The quality assessment becomes more important in recent days, since distributed knowledge emerges rapidly in different application areas. The paper reports the domain of industrial configuration and production, where distributed knowledge bases have been maintained manually over decades. We describe the configuration ontology COOM and show how standard technologies can be used to query experience-based anomalies. A selection of anomalies is discussed.

Keywords: Knowledge Evaluation; Design Anomaly; Ontology; Inspection

1 Introduction

The ongoing automation in the industrial domain enabled the creation of new consumer goods and processes. The increased automation of processes for recommendation, configuration, and production forced a number of periphery procedures to be automated as well. In current state-of-the-art settings, large knowledge bases are executed to run the underlying processes. These knowledge is located in different industrial systems, ranging from sales systems to standard PLM (product life-cycle system) and ERP (enterprise resource planning) systems.

The safe execution of the knowledge need appropriate quality assessment methods. Quality assessment research proposed well-known methods for knowledge validation and verification, e.g. [Pr94, SK94, WL02, Ba11, PS94, VC99, BC99]. Also, quality assessment was investigated in the context of ontologies [Ko14, Vr10, La17]. Besides these well-known assessment areas, we see *design anomalies* as a class that especially focuses on the sustainable development of knowledge bases: In knowledge bases, design anomalies [BS10] identify areas of the knowledge base that do not only cause dysfunction of the knowledge, but also may yield weak maintainability and analysis capabilities.

In the domain of industrial information systems, the quality assessment of *distributed knowledge bases* becomes even more important: Here, knowledge about the same objects can be found almost always in different systems. For example, for a bike manufacturer knowledge about a brake or a gear-box can be found in a sales system, the PLM, and the

[1] Universität Würzburg, Germany joba@uni-wuerzburg.de

ERP system. The consistent analysis and assessment of this distributed knowledge base is a difficult and yet unsolved industrial problem.

In this paper, we first introduce a general ontology for representing industrial artifacts. We then sketch methods for the distributed quality assessment focusing on design anomalies of industrial knowledge. Typically, these methods are based on human experience. For this reason, we describe an approach for the declarative definition of new anomalies. The approach is demonstrated by an exemplary application. We also report a reference implementation of the COOM ontology and the described anomalies. The paper concludes with a summary and a discussion of related work.

2 The COOM Ontology and Linking Distributed Knowledge

In this section, we introduce the COOM ontology. The Configuration Objects Ontology Model (COOM) defines a general description of industrial products, features, and combining/constraining knowledge. It was originally designed to be used for the formulation of configuration knowledge, but can be also used in sales and production processes. Since the introduction of the entire ontology model would exceed the size of this paper, we will focus on the most important concepts. We exemplify the introduced concepts by the running example *dBike*, which is a virtual bike manufacturer. Here, customers can configure their desired bicycles from a range of configuration items.

As depicted in Figure 1 the ontology defines typical artifacts of a producing company:

- products and their structure
- features and feature items of products
- relational knowledge for customization and configuration

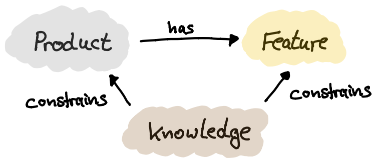


Fig. 1: The COOM triangle of products, features, and knowledge.

For the definition of the ontology we extend the SKOS ontology [W309], which already defines a number of basic concepts for general knowledge organization systems. The central class `LinkedConcept` is derived from `skos:Concept` and describes the linked characteristics of concepts between a number of information systems. Instances of LinkedConcept own properties `linkedSystem` pointing to the original information system and `linkedId` storing the original identifier of the concepts. Both relations are necessary to obtain a standardized link to the originating information system.

2.1 Product Portfolio

All produced artifacts are organized in a hierarchical product portfolio as seen in Figure 2. The SKOS concept scheme is applied to define the product portfolio: A hierarchy is formed

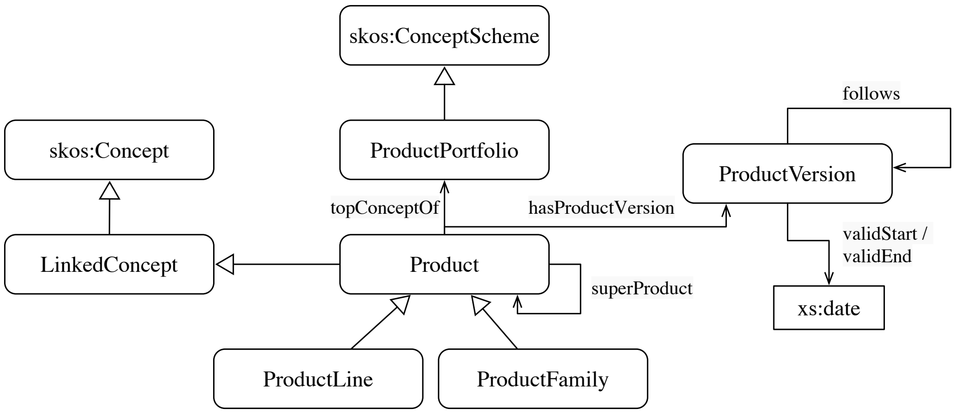


Fig. 2: Classes and properties of a generic product portfolio.

by instances of `Product` connected by `superProduct` relations, which are derivations of `skos:broader`. The hierarchy of the portfolio follows the composite pattern. Special types of products are represented by model families (`ProductFamily`) and subsequent model lines (`ProductLine`). For example, the product portfolio of the dBike company partitions their products into the product families ”Mountain Bike”, ”City Bike”, and ”Racing Bike”. Due to the small product portfolio of the company, there exist no further specialization into product lines. Thus, the product ”dB Racing” is connected by the ”superProduct” relation to the product family ”Racing Bike”.

Different model years of a product are represented in product versions. Instances of `ProductVersion` are connected with a concrete product and a valid start and end date, that defines the respective life cycle of the product. Here, the particular calendar years are the valid dates for the product versions. For example, we introduce a product version ”dBR 2019” for the product ”dB Racing” with valid start date ”2019-01-01” and end date ”2019-12-31”.

2.2 Features, Assignments and Configurations

The ontology was originally defined for the implementation of configuration tasks. Therefore, the COOM ontology defines `Feature` instances for products to represent its specific characteristics. Since features can change over different product versions, a feature is not connected to a concrete product, but to a product versions. Thus, a concrete product version defines a collection of features, that are available in this version. The structure of features

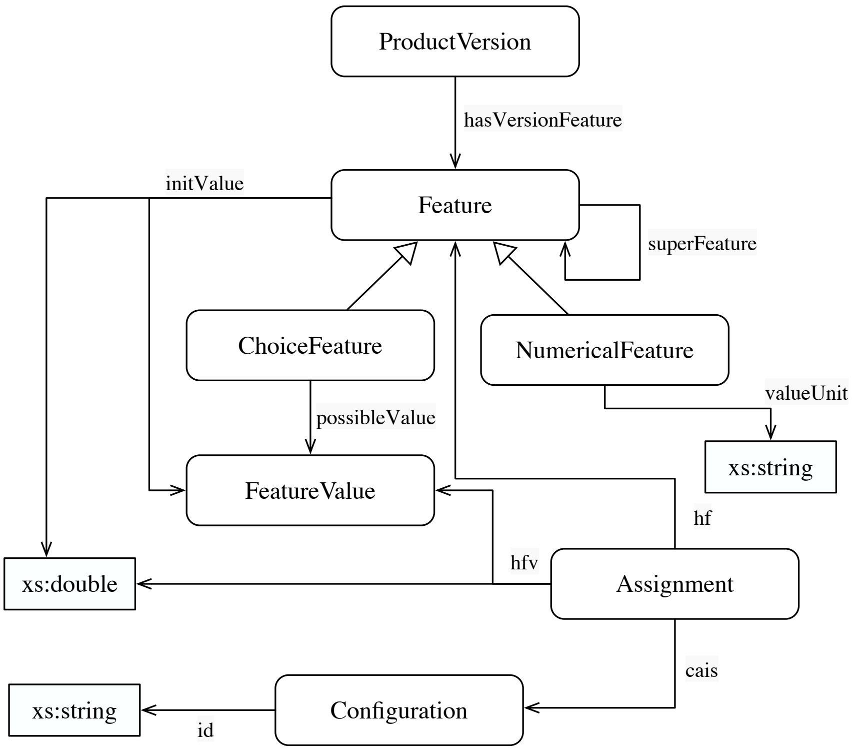


Fig. 3: Classes and properties defining the features, an assignment to feature values, and a configuration.

is shown in Figure 3. The particular features are organized hierarchically and concrete feature instances are connected by the `superFeature` property, a derivation of `skos:broader`. Different types of features are characterized by the value type it can be assigned to. We define `NumericalFeature` storing float values (e.g., length of a product) and `ChoiceFeature` having a predefined list of possible choice values (e.g., color of a product). Consequently, we introduce properties for assigned choice values to Features (`possibleValue`) and initial values (`initValue`, sometimes used as defaults). For choice values it is possible to state an order between the different values by the property `valueOrder`, e.g., for a feature ”gear” the ordered choices ”7G”, ”11G”, and ”27G”.

For example, the product version ”dBR 2019” defines the choice feature ”brake” for the racing bike of the year 2019: The feature ”brake” has the possible values ”race brake”, ”standard brake”, and ”heavy duty brake”.

An assignment between a value and a feature is captured by the corresponding class `assignment`, that provides the property `hf` (has feature) to reference the feature instance and the property `hfv` (has feature value) to reference the value. Please note, that the value class need to correspond to the assigned feature class. In our example, a possible assignment is ”brake = race brake”.

A `Configuration` instance collects all feature–value items (assignments) of a specific product. It is worth noticing, that one assignment needs to reference assignment pairs that corresponds to the same product version. Also, a `Configuration` instance usually refers to an identifier that distinguishes it from other configurations. Often, this identifier is called serial number or machine number.

For example, the order of a customer is stored in a configuration instance. This instance collects the assignments ”frame = racing frame”, ”brake = racing brake”, ”gear = 27G”, ”light = superlight”, and ”color = red”.

2.3 Relational Knowledge

As described in the introduction, knowledge constraints products and features. Figure 4 depicts different types of relational knowledge. Constraints on features are represented by `condition` instances, that define the requirements for the execution of this particular knowledge element. The concrete condition vary between the sub-classes of `RelationKnowledge`: For example, forbidden combinations of feature values in a specific product are defined by an `InvalidValue` constraint. Following our example, an invalid value constraint could be defined between the assignments ”frame = racing frame” and ”gear = 7G”, i.e., a 7-gearbox must not be combined with a racing frame.

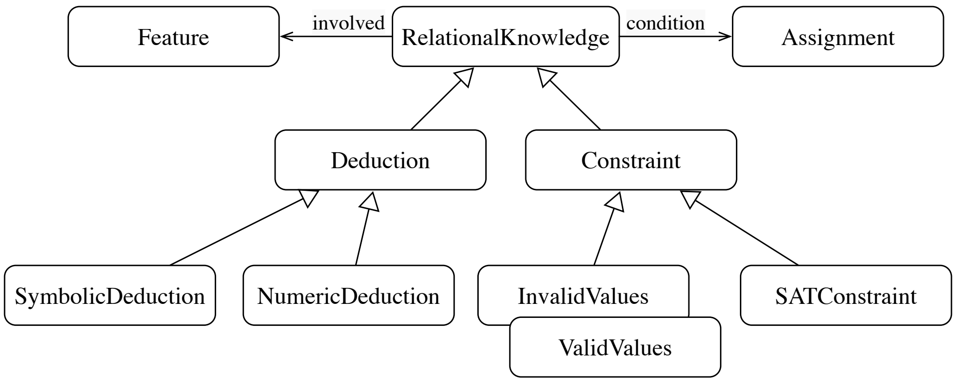


Fig. 4: Classes and properties defining the relational knowledge, mostly between feature values.

Besides Boolean constraints on feature values, there also exists knowledge for deriving specific feature values for a given condition. The derivation of feature values can be defined for choice features (`SymbolicDeduction`) and for numerical features (`Calculation`). For instance, there may exist a symbolic deduction rule, that derives the assignment ”light = superlight” for a given assignement ”frame = racing frame”.

A `SATConstraint` defines a (often complex) condition, that need to be either positively or negatively satisfied. The simplified analysis and exchange of relational knowledge is implemented by the relation `involved`, that connects all participating Features for an instance of `RelationalKnowledge`. We also define the sub-properties `conditionedFeature` and `derivedFeature` for, depending of the specific type of the relational knowledge, the features used in the condition but also in the conclusion are linked by these properties.

The detailed representation of condition and deduction can be implemented by the standard the rule language SWRL [Ho04]. Albeit the syntax of SWRL is not easy to comprehend for untrained users, it provides a common standard for representing equations and conditions. In the context of our work, however, we focus on the shallow representation of conditioned features and derived features, since this knowledge is easy to interchange between systems and is sufficient to answer a number of analysis and evaluation questions.

2.4 Linking Distributed Knowledge

In the previous sections we introduced the general scheme of the COOM ontology. The knowledge included in the particular systems of the company can be mapped to a instance of the COOM scheme.

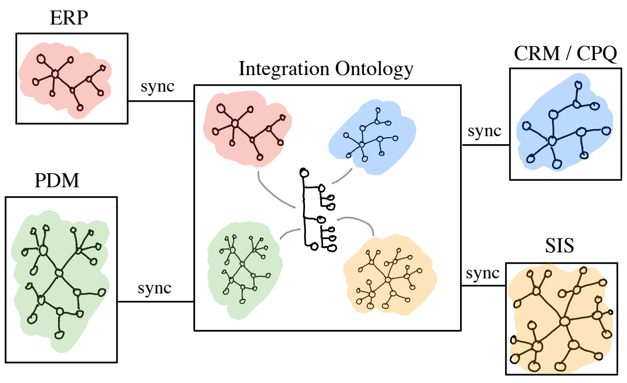


Fig. 5: Linking knowledge of existing information systems with an integration knowledge base.

Figure 5 shows the different knowledge bases contained in system instances of ERP, PDM, CRM and SIS applications. In this section, we propose an approach, where these local knowledge bases are connected by an integration ontology. In consequence, the integration ontology can be used for all analysis and evaluation tasks.

In an exemplary situation, a company runs a product data management system (PDM) in the engineering department, a customer relations and pricing system in the sales department (CRM/CPQ), and a service information system (SIS) in the after sales department. Also an enterprise resource planing system (ERP) is connected for managing the master data of the company. The knowledge included in these systems considers the same products, features, and interrelations.

In a canonical mapping scheme we link all elements to the integration ontology together with their local namespaces of each originating system. For example, a specific product ”dB Racing” and its instance `dBR`, respectively, is referenced in all information systems. We also introduce the corresponding instances `erp:dBR`, `pdm:dBR`, `cpq:dBR`, and `sis:dBR` of `Product`. The identity semantics between all dBR instances is represented by the sub-properties of `coom:match` that is a sub-property of `skos:mappingRelation` [W309]. In consequence, we arrive at one integration ontology and one ontology for each connected information system.

The sketched approach produces redundant instances for each concept, but allows for an independent development and use of the different knowledge systems. Also, we can simply combine different versions of each knowledge base and information system, respectively.

For the quality assessment of the distributed knowledge base, it is not necessary to include and map the complete depth of the knowledge base. It is rather necessary to map levels of knowledge that are used in the evaluation task. We show examples of quality assessment methods in the following section.

3 Experience-based Quality Assessment of COOM Knowledge Graphs

As we motivated in the previous sections, there will not exist a single knowledge model in a typical industrial setting. Rather, each installed information system (ERP, PLM, CRM, etc.) will contain knowledge that can be mapped to a separate knowledge model. In Figure 5 we sketch a common structure of such a *distributed knowledge model*.

In this paper, we focus on quality assessment methods that consider the maintainable design of distributed knowledge models, i.e., *design anomalies*. A very simple measure of such a design anomaly is the compliance with defined naming conventions for resource names. More sophisticated measures try to identify unused or misused elements in the knowledge graph. The idea of design anomalies is closely related to *ontology anti-patterns* [CRVB09].

It should be clear, that there will be no exhaustive list of methods but that with the particularities of each domain and the experience of the knowledge engineers, there will always emerge new measures to be implemented in the quality assessment framework. For example, with new system capabilities new quality demands will rise. Therefore, we propose an experience-based approach to formulate assessment methods. In the best case, new measures can be implemented in a declarative manner. In the past, semantic languages were defined for implementing quality measure, such as SPARQL, ShEx and SHAQL, see for instance [La17, FH10]. In the following, we introduce a series of methods that are helpful for the distributed development of knowledge models, and we use these languages whenever possible.

3.1 Lonely Feature

A feature should be affiliated with at least one knowledge model (sales model, engineering model, etc.). The following query looks for lonely features that are not connected to any model.

***Anomaly* 1 (Lonely Feature)** *A feature f is called a **lonely feature**, when there exists no known knowledge model, that includes this feature.*

The following SPARQL query reports all lonely features in the environment of the SPARQL query.

```
# Lonely Feature
SELECT ?feature_uri
WHERE {
  ?feature_uri a coom:Feature .
  MINUS { ?feature_uri coom:inModel ?model }
}
```

3.2 Childless Feature

A *childless feature* is detected for a choice feature, that does not define at least one feature value. In more restrictive tests, every choice feature need to define more than one feature value.

***Anomaly* 2 (Childless Feature)** *We introduce a choice feature f with $type(f) = discrete$ and $dom(f)$ yields the domain of feature f, i.e., the possible values of the feature. The feature f is a **childless feature**, if $f \in M \wedge dom(f) = \emptyset$.*

A childless feature signals an orphan resource in the knowledge model, when a knowledge engineer stopped working on a topic and forgot to remove unnecessary resources from the model. Another explanation is the unfinished type change of resource, e.g. moving a numeric feature to a choice feature or counter-wise.

The following SPARQL query reports all childless features.

```
# Childless Feature
SELECT ?f ?value
WHERE { ?f a coom:ChoiceFeature ;
        FILTER NOT EXISTS {
           ?f coom:possibleValue ?value }
}
```

3.3 Uneven Twins

The measure *uneven twins* tries to find two matching features from different knowledge models, for which at least one feature value has no matching counterpart for the values of the other feature. In practice, the knowledge engineers may forgot to define a match relation between feature values. Another reason for an uneven twin can be the change of the semantics of values without propagating this to the other information systems.

***Anomaly* 3 (Uneven Twins)** *We introduce two matching features f_1 and f_2 that are included in two different knowledge models M_1 and M_2:*

$$f_1 \in M_1, f_2 \in M_2 : match(f_1, f_2) .$$

Both features f_i have discrete values $v_{i,j}$ defined in the domain $dom(f_i)$:

$$dom(f_1) = \{v_{1,1}, v_{1,2}, \ldots, v_{1,n}\} \text{ and } dom(f_2) = \{v_{2,1}, v_{2,2}, \ldots, v_{2,m}\} .$$

*The features f_1, f_2 are **uneven twins**, if there exists a feature value $v \in dom(f_1)$ but no corresponding value $v' \in dom(f_2)$ with the matching relation match(v, v').*

The following SPARQL statement shows a possible query for identifying uneven twins `f1` and `f2`.

```
# Uneven Twins
SELECT  ?f1 ?f1_value ?model
WHERE { ?f1 a coom:Feature ;
            coom:possibleValue ?f1_value ;
            coom:inModel ?model .
            MINUS { ?f2 a coom:Feature ;
                        coom:possibleValue ?f2_value .
                    ?f1 coom:match ?f2 .
                    ?f1_value coom:match ?f2_value .
                    FILTER (?f1 != ?f2) }
}
```

3.4 Knowledge Twins

Distributed knowledge models often yield similar or even equal knowledge elements in different knowledge systems. For instance, the sales model may introduce the same constraint as the engineering model did before. The anomaly knowledge twin tries to identify such *doublettes*. After a detection, a human knowledge engineer needs to decide about how to handle identified twins.

***Anomaly* 4 (Knowledge Twins)** *We define **knowledge twins** as two different knowledge elements k_1 and k_2 ($k_1 \neq k_2$), that derive feature values v for the same features f with same or intersecting feature sets in the condition, i.e.,*

$$k_1 : \underbrace{\{c_1, \ldots, c_n\}}_{C} \rightarrow \underbrace{\{a_1, \ldots, a_m\}}_{A} \quad \wedge \quad k_2 : \underbrace{\{c'_1, \ldots, c'_p\}}_{C'} \rightarrow \underbrace{\{a'_1, \ldots, a'_q\}}_{A'},$$

where c_i, c'_i, a_i, a'_i are assignments $f = v$ of values $v \in dom(f)$ to features f. Two sets of assignments C, C' are intersecting, *when there exists at least one assignment in each set, that have matching features and feature values, i.e.,*

$$c \in C, a \in A, c' \in C', a' \in A' \ : \ match(c, c') \wedge match(a, a') .$$

Please notice, that in the implementation the property `coom:match` is reflexive and thus a feature also has an exact match to itself.

The following SPARQL statement shows a simplified query for detecting knowledge twins `k1` and `k2` in different knowledge models.

```
# Knowledge Twins
SELECT  ?k1 ?k2
WHERE { ?k1 a coom:RelationalKnowledge ;
            coom:conditionedFeature ?f1con ;
            coom:derivingFeature ?f1der ;
            coom:inModel ?model1 .
        ?k2 a coom:RelationalKnowledge ;
            coom:conditionedFeature ?f2con ;
            coom:derivingFeature ?f2der ;
            coom:inModel ?model2 .
        FILTER (?k1 != ?k2)
        FILTER (?model1 != ?model2)
        FILTER EXISTS {
           ?f1con coom:match ?f2con .
           ?f1der coom:match ?f2der . }
}
```

3.5 Incompatible Concept Matching

An incompatible concept matching is found for two features, that are defined to match but have different information types. Incompatible concept matching often occurs in the progress of restructurings of larger distributed knowledge systems: The type of a feature was modified in one model due to a design decision but the change was not propagated to the other knowledge models.

***Anomaly* 5 (Incompatible Concept Matching)** *We define two knowledge models M_1, M_2 having features $f_1 \in M_1$ and $f_2 \in M_2$. There exists an **incompatible concept matching**, if both features are matching, i.e., $match(f_1, f_2)$ but have different information types, i.e., $type(f_1) \neq type(f_2)$.*

The following SPARQL queries for a feature `f1` that has an exact match to a feature `f2` with class `f2Type` that is different from all type classes of `f1`.

```
# Incompatible Concept Matching
SELECT  ?f1
WHERE { ?f1 a coom:Feature ;
            coom:match/rdf:type ?f2Type ;
        MINUS { ?f1 a ?f2Type . }
}
```

3.6 Similar Surface

During the distributed development the definition of matching relations may be incomplete. We introduce a very shallow and simple anomaly, that (nevertheless) is very helpful to exploit many missing matching relations. A similar surface for two features exists, when both features have similar/same names but a matching relation is missing. It is obvious, that a possible matching relation needs to be inserted manually by a knowledge engineer after an inspection of the anomaly.

***Anomaly* 6 (Similar Surface)** *We define two knowledge models M_1, M_2 having features $f_1 \in M_1$ and $f_2 \in M_2$. There exists a **similar surface** for f_1 and f_2, if*

$$\neg match(f_1, f_2) \wedge similar(f_1, f_2) .$$

The implementation of the function *similar* can vary from a very simple string comparison of the resources labels to a sophisticated resource matching algorithm. The ontology matching research offers a diverse range of methods that should be considered for the implementation of this anomaly measure [ES12].

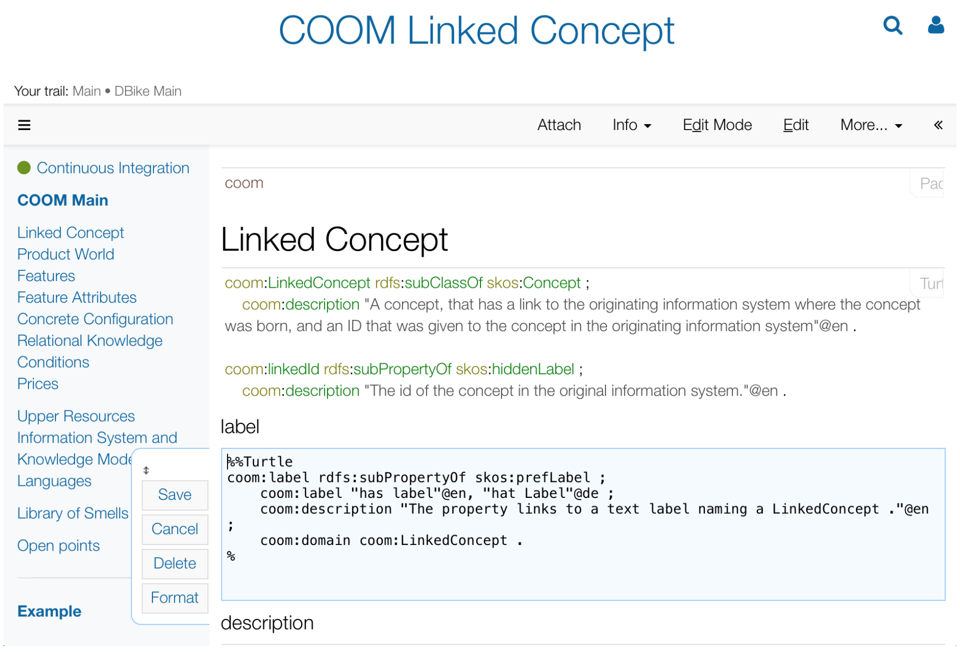
The following SPARQL statement implements a simple version, where we query for features `f1` and `f2` that have an identical label literal.

```
# Similar Surface
SELECT ?f1 ?f2
WHERE {  ?f1 a coom:Feature ;
             coom:label ?label .
         ?f2 a coom:Feature ;
             coom:label ?label .
         FILTER (?f1 != ?f2)
         FILTER NOT EXISTS { ?f1 coom:match ?f2 }
}
```

4 Implementation

We implemented the described COOM ontology using the knowledge engineering tool KnowWE [BRP11]. The application KnowWE is a semantic wiki, that supports the distributed elicitation and maintenance of RDF(S)/OWL ontologies. It provides mechanisms to import existing ontologies and offers markups to create new ontologies by using Turtle syntax. Graphs can be queried with SPARQL statements. Ontology statements in Turtle markup are directly compiled into an ontology and can be accessed by inserted SPARQL queries, also via a web-service endpoint.

Figure 6 shows a part of the implementation, i.e., the KnowWE page of `LinkedConcept` with an open editor defining the related property `coom:label`. Among other features, KnowWE offers a rich set of tools for evaluating the knowledge bases. That way, various test types can be defined to inspect engineered ontologies. For a sustainable quality assessment a continuous integration dashboard is integrated into KnowWE [BR11]. This dashboard runs a defined suite of tests every time, the knowledge has been changed. Detected errors or warnings are reported visually to the user. On the top left corner of Figure 6, we can see a green circle (followed by a link to "Continuous Integration"). In case of problems, this circle turns red and the user can inspect the dashboard report by just clicking on the corresponding link. In the context of this paper, we implemented—among others—the described anomalies as named SPARQL queries. Named SPARQL queries are easily added as a test into the test suite of the dashboard, for instance by demanding that the query should yield zero results. By using a semantic wiki and a standard query language for the definition of anomalies, new experience knowledge with respect to quality assessment can be easily added to the system.



coom:LinkedConcept rdfs:subClassOf skos:Concept ;
    coom:description "A concept, that has a link to the originating information system where the concept was born, and an ID that was given to the concept in the originating information system"@en .

coom:linkedId rdfs:subPropertyOf skos:hiddenLabel ;
    coom:description "The id of the concept in the original information system."@en .

```
%%Turtle
coom:label rdfs:subPropertyOf skos:prefLabel ;
    coom:label "has label"@en, "hat Label"@de ;
    coom:description "The property links to a text label naming a LinkedConcept ."@en
;
    coom:domain coom:LinkedConcept .
%
```

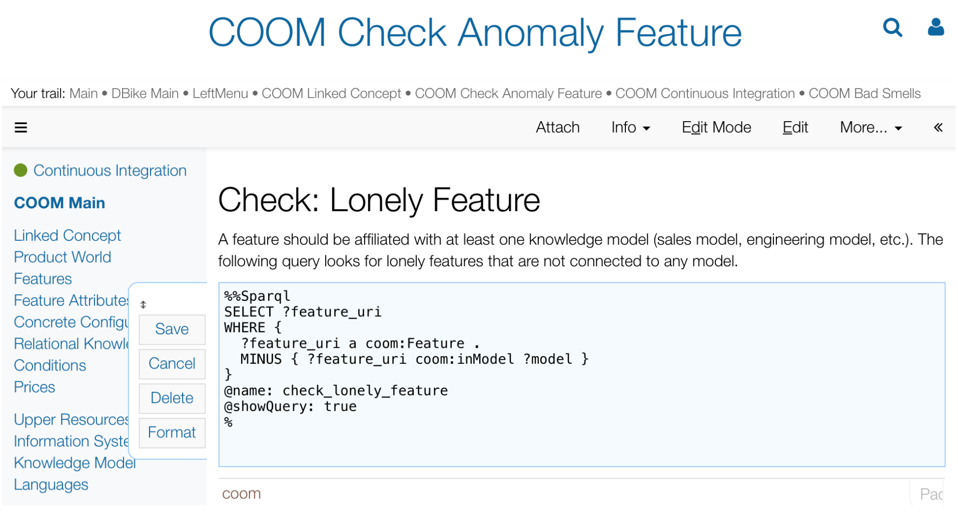
Fig. 6: The knowledge engineering tool KnowWE depicting the page for `LinkedConcept`. The turtle editor is open for property `coom:label`.

In Figure 7 we see the definition of the anomaly Lonely Feature as a named SPARQL query. The query is used in the definition of the dashboard in Figure 8 demanding that there should be zero results for this query.

5 Conclusions

The quality assessment of distributed knowledge bases has many application areas. In this paper, we introduced configuration and production of industrial goods as an interesting application domain, where distributed knowledge bases are already developed and maintained over the past decades.

We introduced the ontology schema COOM (Configuration Objects Ontology Model), that builds on the standard knowledge organization system SKOS and is itself extensible. Furthermore, we showed how COOM can be implemented in a distributed knowledge ecosystem. In the past, general approaches of knowledge-based configuration [Fe14] were introduced. In comparison, the proposed ontology introduces concepts and constraint types that are tailored to the use in the manufacturing industry and sketches an approach of distributed quality assessment.



Check: Lonely Feature

A feature should be affiliated with at least one knowledge model (sales model, engineering model, etc.). The following query looks for lonely features that are not connected to any model.

```
%%Sparql
SELECT ?feature_uri
WHERE {
  ?feature_uri a coom:Feature .
  MINUS { ?feature_uri coom:inModel ?model }
}
@name: check_lonely_feature
@showQuery: true
%
```

Fig. 7: Definition of anomaly Lonely Feature.

Experience-based evaluation of such knowledge bases becomes more important these days. We introduced a number of design anomalies, that can be implemented by standard technologies, such as SPARQL, ShEx, and SHAQL. By using these standard languages, the declarative extension of an experience-based anomaly library is a feasible task. This is similar to previous works. For instance, in [Ro12] the authors use SPARQL queries to detect anti-patterns.

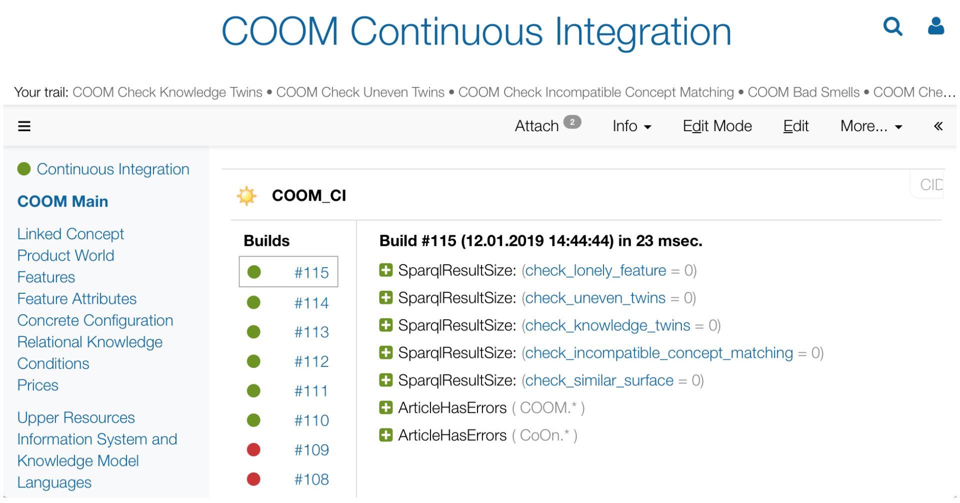
At the moment, the implementation in KnowWE only considers tests defined by SPARQL queries. Albeit very powerful, this mechanism is burdensome for creating many (rather simple) anomalies. Therefore, we are planning to integrate an implementation of ShEx and SHAQL into KnowWE and its quality dashboard in the future. We expect to simplify the definition of experience-based evaluation knowledge even further.

Furthermore, we are planning to implement a more comprehensive library of anomalies based on the possibilities of ShEx, SHAQL, and SPARQL. Having a representative set of anomalies with a reference implementation we will be able to gain more public interest.

References

[Ba11] Baumeister, Joachim: Advanced Empirical Testing. Knowledge-Based Systems, 24(1):83–94, 2011.

[BC99] Boswell, Robin; Craw, Susan: Organizing Knowledge Refinement Operators. In: Validation and Verification of Knowledge Based Systems. Kluwer, Oslo, Norway, pp. 149–161, 1999.


COOM Continuous Integration

Your trail: COOM Check Knowledge Twins • COOM Check Uneven Twins • COOM Check Incompatible Concept Matching • COOM Bad Smells • COOM Che...

● Continuous Integration

COOM Main

Linked Concept
Product World
Features
Feature Attributes
Concrete Configuration
Relational Knowledge
Conditions
Prices

Upper Resources
Information System and Knowledge Model
Languages

COOM_CI

Builds	
🟢	#115
🟢	#114
🟢	#113
🟢	#112
🟢	#111
🟢	#110
🔴	#109
🔴	#108

Build #115 (12.01.2019 14:44:44) in 23 msec.

SparqlResultSize: (check_lonely_feature = 0)
SparqlResultSize: (check_uneven_twins = 0)
SparqlResultSize: (check_knowledge_twins = 0)
SparqlResultSize: (check_incompatible_concept_matching = 0)
SparqlResultSize: (check_similar_surface = 0)
ArticleHasErrors (COOM.*)
ArticleHasErrors (CoOn.*)

Fig. 8: The continuous integration dashboard of the semantic wiki KnowWE.

[BR11] Baumeister, Joachim; Reutelshoefer, Jochen: Developing Knowledge Systems with Continuous Integration. In: i-KNOW 2011: 11th International Conference on Knowledge Management and Knowledge Technologies, short paper. ACM ICPS, Graz, Austria, 2011.

[BRP11] Baumeister, Joachim; Reutelshoefer, Jochen; Puppe, Frank: KnowWE: A Semantic Wiki for Knowledge Engineering. Applied Intelligence, 35(3):323–344, 2011.

[BS10] Baumeister, Joachim; Seipel, Dietmar: Anomalies in Ontologies with Rules. Web Semantics: Science, Services and Agents on the World Wide Web, 8(1):55–68, 2010.

[CRVB09] Corcho, Oscar; Roussey, Catherine; Vilches Blazquez, Luis Manuel: Catalogue of Anti-Patterns for formal Ontology debugging. In: Atelier Construction d ontologies : vers un guide des bonnes pratiques, AFIA 2009. Hammamet, Tunisia, p. 11, May 2009.

[ES12] Euzenat, Jérôme; Shvaiko, Pavel: Ontology Matching. Springer, Berlin, 2nd edition, 2012.

[Fe14] Felfernig, Alexander; Hotz, Lothar; Bagley, Claire; Tiihonen, Juha: Knowledge-based Configuration: From Research to Business Cases. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition, 2014.

[FH10] Fürber, Christian; Hepp, Martin: Using SPARQL and SPIN for Data Quality Management on the Semantic Web. In (Abramowicz, Witold; Tolksdorf, Robert, eds): Business Information Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 35–46, 2010.

[Ho04] Horrocks, Ian; Patel-Schneider, Peter F.; Boley, Harold; Tabet, Said; Grosof, Benjamin; Dean, Mike: , SWRL: A Semantic Web Rule Language - Combining OWL and RuleML, W3C Member Submission . `http://www.w3.org/Submission/SWRL/`, May 2004.

[Ko14] Kontokostas, Dimitris; Westphal, Patrick; Auer, Sören; Hellmann, Sebastian; Lehmann, Jens; Cornelissen, Roland; Zaveri, Amrapali: Test-driven Evaluation of Linked Data

Quality. In: Proceedings of the 23rd International Conference on World Wide Web. WWW ’14, ACM, New York, NY, USA, pp. 747–758, 2014.

[La17] Labra Gayo, Jose Emilio; Prud'hommeaux, Eric; Boneva, Iovka; Kontokostas, Dimitris: Validating RDF Data. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers LLC, sep 2017.

[Pr94] Preece, Alun D: Validation of Knowledge-Based Systems: The State-of-the-Art in North America. The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistomology, 11, 1994.

[PS94] Preece, Alun; Shinghal, Rajjan: Foundation and Application of Knowledge Base Verification. International Journal of Intelligent Systems, 9:683–702, 1994.

[Ro12] Roussey, Catherine; Corcho, Oscar; Šváb Zamazal, Ondřej; Scharffe, François; Bernard, Stephan: SPARQL-DL Queries for Antipattern Detection. In: Proceedings of the 3rd International Conference on Ontology Patterns - Volume 929. WOP’12, CEUR-WS.org, Aachen, Germany, Germany, pp. 85–96, 2012.

[SK94] Smith, Suzanne; Kandel, Abraham: Verification and Validation of Rule-Based Expert Systems. CRC Press, Inc., Boca Raton, FL, USA, 1994.

[VC99] Vermesan, Anca; Coenen, Frans: Validation and Verification of Knowledge Based Systems. Theory, Tools and Practice. Kluwer Academic Publisher, 1999.

[Vr10] Vrandecić, Denny: Ontology Evaluation. PhD thesis, AIFB, KIT Karlsruhe, Germany, 2010.

[W309] W3C: , SKOS Simple Knowledge Organization System Reference: `http://www.w3.org/TR/skos-reference`, August 2009.

[WL02] Wu, Chih-Hung; Lee, Shie-Jue: KJ3 – a tool assisting formal validation of knowledge-based systems. International Journal of Human-Computer Studies, 56(5):495–524, 2002.