Virtual Properties for Ontologies and Knowledge Graphs

Henrik Dibowski [1]

[1] *Robert Bosch GmbH, Bosch Center for Artificial Intelligence, 71272 Renningen, Germany*

Abstract

This paper presents virtual properties for ontologies and knowledge graphs (KGs). Virtual properties are virtually computed shortcuts on a KG connecting information at distant places of a KG, reachable via object property paths, directly to a given class. Virtual properties are a valuable UI concept, which can make a KG much better viewable, readable and searchable. In a UI that provides a frame-based view on a KG, virtual properties can be shown additionally to the "normal", direct properties of a class. This can significantly improve the content and value of the UI by enriching the scope of a class with other information beyond its direct neighborhood.

Virtual properties can be defined as enhanced, repurposed SHACL property shapes, as part of the ontology schema. The values of virtual properties can be computed on-demand with dedicated SPARQL queries, or alternatively rules, which resolve their defined object property path in the KG at runtime. Virtual properties can be realized as integral functionality of generic, frame-based UIs, which can automatically generate views and masks for viewing and searching a KG, comprising both virtual and non-virtual properties of a class. They can be included in the definition of search queries, which enables an advanced KG search far beyond the direct scope of a class. Virtual properties support multilinguality in their names, paths and values (in case of string datatype properties) by using language-tagged string values.

The virtual property approach has been implemented at Bosch and is being used by currently 100 engineers in a productive setting of a frame-based UI. It has successfully demonstrated the feasibility and significant value that virtual properties can provide.

Keywords

Virtual Property, Property Path, Knowledge Graph Search, Ontology, SHACL, SPARQL

1. Introduction

Knowledge graphs (KGs) are on the rise and are spreading into more and more industrial use cases. Their strength of representing heterogeneous, highly connected information, of making it accessible via semantic search, and of inferring new facts via reasoning is superior to other existing data representations and data storage solutions.

A crucial, still limiting factor, are adequate means for easily viewing, browsing and searching the contents of KGs. Software tools need to ensure a high user experience and efficiency for increasing the acceptance and success of KGs. Also, non-experts, with limited background in KGs and ontologies need to be able to use such tools intuitively and efficiently, which is rarely the case. A particular challenge of KGs consists in their much more distributed, heterogeneous structure, with many nodes and relations, which is generally more difficult to visualize. Basic table views like in relational databases or spreadsheets that most users are used to, are not suitable for KGs. Instead, frame-based UIs and graph visualizations have become the main concepts for viewing and browsing KGs. But both have limiting shortcomings, that will be discussed in Section 2.

IWMSW-2022: International Workshop on Multilingual Semantic Web, Co-located with the KGSWC-2022, November 21–23, 2022, Madrid, Spain.
EMAIL: henrik.dibowski@de.bosch.com
ORCID: 0000-0002-9672-2387

As a main contribution of this paper, the concept of *virtual properties* is introduced in this paper. Virtual properties can improve the viewing, browsing and searching of KGs, and can contribute to a better user experience and acceptance of KGs.

The paper is structured as follows: Section 2 describes the available means for viewing and browsing KGs, and their shortcomings. Section 3 is the main part of the paper and introduces virtual properties in detail. Section 4 discusses the differences between virtual properties and OWL 2 property chains. An evaluation is given in Section 5, followed by the conclusion in Section 6.

2. Related work

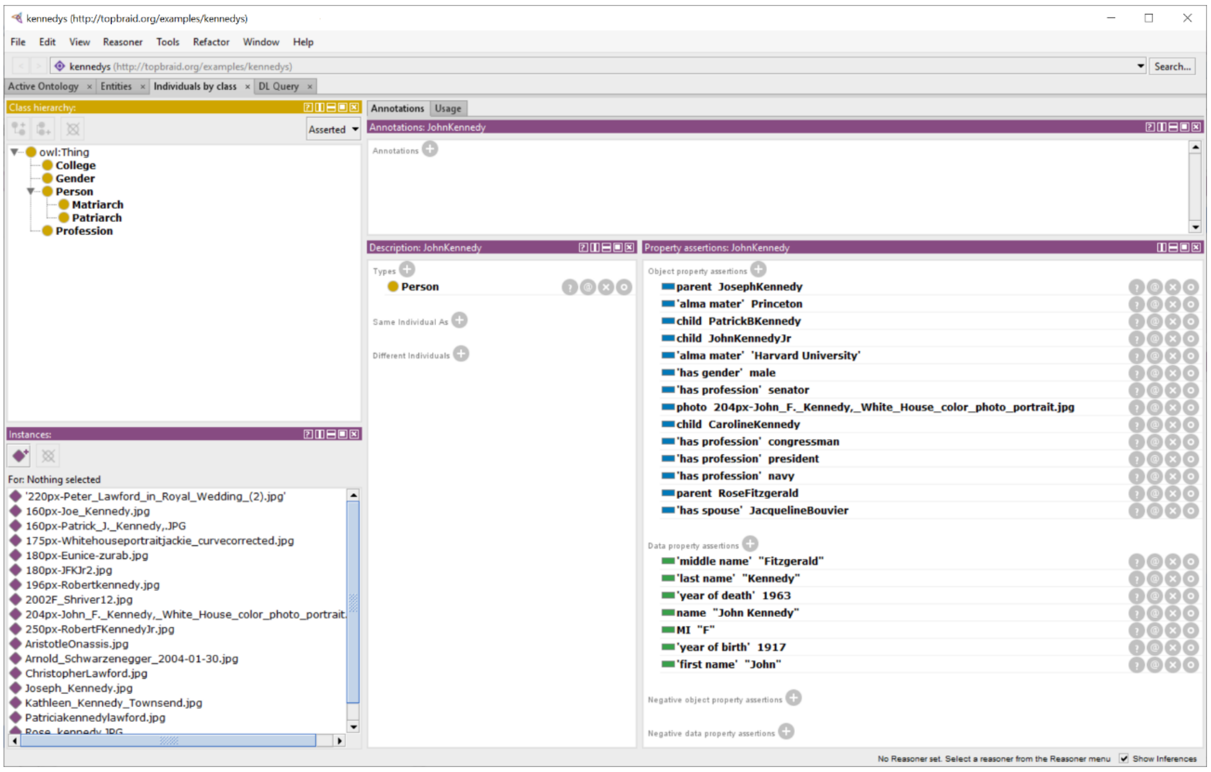
There are mainly two approaches for viewing and browsing KGs: *Frame-based views* and *graph visualizations*.

Frame-based views on KGs are predominant in many ontology IDEs, such as Protégé [1], WebProtégé [2] or TopBraid Composer [3]. For a comparison, the Kennedys ontology of TopQuadrant [4] is shown in Protégé in Figure 1, and in TopBraid Composer in Figure 2. Both tools provide an instance-centric view on the KG, in which a single instance with all its defined properties and values is shown. The advantage of frame-based views is that they offer a good overview of all properties of an instance at a glance, and they are intuitive to use by most of the users. Also, they are well suited for expanding and changing ontologies and KGs. Such instance-centric views however can only show a very limited subset of the KG at a time, namely one instance and its direct properties. Seeing additional information requires the user to navigate along the object properties to other connected instances to see and understand that part of the KG.

Graph visualizations try to show subsets of a KG as graph, using different layouts and designs. Examples are VOWL [5], OWLViz[1], OntoGraf[2] and Stardog Explorer [6]. As one representative example, the visualization of Stardog Explorer is shown in Figure 3. The nodes in the graph represent either classes or instances, and the edges properties. Such graphs are not limited to showing only one instance with its properties, but they can show multiple instances and their properties at a time. They are good means for creating aha effects, when showing them to people. Their clear disadvantage is that they are only readable with a quite low number of nodes and edges. Very quickly they get overcrowded, difficult to view and browse. Additionally, achieving a good layout is difficult and usually the graph views are nondeterministic, meaning that visualizing the same subset of a KG the next time, will result in a totally different layout. Expanding KGs with graphical visualizations is possible with some tools, but often difficult, due to the afore mentioned problems.

As a conclusion, neither frame-based views nor graph visualizations are optimal for viewing, browsing or searching KGs. A solution that can overcome the limitations of frame-based views are virtual properties. The core concept of virtual properties has first been introduced in [7]. The concept of virtual properties has been widely enhanced and will be explained in detail in the following.

[1] https://github.com/protegeproject/owlviz
[2] https://github.com/protegeproject/ontograf



kennedys (http://topbraid.org/examples/kennedys)
File Edit View Reasoner Tools Refactor Window Help
kennedys (http://topbraid.org/examples/kennedys)
Search...
Active Ontology
Entities
Individuals by class
DL Query
Class hierarchy:
Asserted
Annotations
Usage
Annotations: JohnKennedy
Annotations
Description: JohnKennedy
Types
Same Individual As
Different Individuals
Property assertions: JohnKennedy
Object property assertions
Data property assertions
Negative object property assertions
Negative data property assertions
Instances:
For: Nothing selected
No Reasoner set. Select a reasoner from the Reasoner menu
Show Inferences

- owl:Thing
  - College
  - Gender
  - Person
    - Matriarch
    - Patriarch
  - Profession

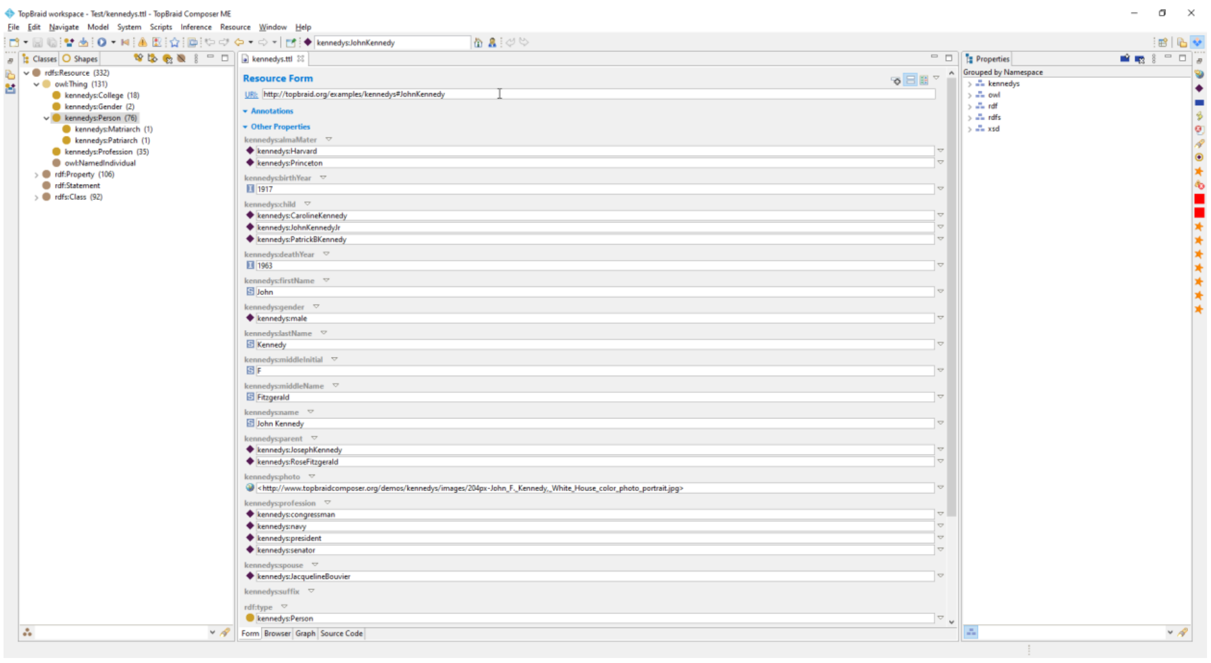
Types: Person

Object property assertions:
- parent JosephKennedy
- 'alma mater' Princeton
- child PatrickBKennedy
- child JohnKennedyJr
- 'alma mater' 'Harvard University'
- 'has gender' male
- 'has profession' senator
- photo 204px-John_F._Kennedy,_White_House_color_photo_portrait.jpg
- child CarolineKennedy
- 'has profession' congressman
- 'has profession' president
- 'has profession' navy
- parent RoseFitzgerald
- 'has spouse' JacquelineBouvier

Data property assertions:
- 'middle name' "Fitzgerald"
- 'last name' "Kennedy"
- 'year of death' 1963
- name "John Kennedy"
- MI "F"
- 'year of birth' 1917
- 'first name' "John"

Instances:
- '220px-Peter_Lawford_in_Royal_Wedding_(2).jpg'
- 160px-Joe_Kennedy.jpg
- 160px-Patrick_J._Kennedy,.JPG
- 175px-Whitehouseportraitjackie_curvecorrected.jpg
- 180px-Eunice-zurab.jpg
- 180px-JFKJr2.jpg
- 196px-Robertkennedy.jpg
- 2002F_Shriver12.jpg
- 204px-John_F._Kennedy,_White_House_color_photo_portrait
- 250px-RobertFKennedyJr.jpg
- AristotleOnassis.jpg
- Arnold_Schwarzenegger_2004-01-30.jpg
- ChristopherLawford.jpg
- Joseph_Kennedy.jpg
- Kathleen_Kennedy_Townsend.jpg
- Patriciakennedylawford.jpg
- Rose_kennedy.JPG

Figure 1: Frame-based instance view in Protégé on the example of the Kennedys ontology



- rdfs:Resource (332)
  - owl:Thing (131)
    - kennedys:College (18)
    - kennedys:Gender (2)
    - kennedys:Person (76)
      - kennedys:Matriarch (1)
      - kennedys:Patriarch (1)
    - kennedys:Profession (35)
    - owl:NamedIndividual
  - rdf:Property (106)
  - rdf:Statement
  - rdfs:Class (92)

URI: http://topbraid.org/examples/kennedys#JohnKennedy

kennedys:almaMater	kennedys:Harvard
	kennedys:Princeton
kennedys:birthYear	1917
kennedys:child	kennedys:CarolineKennedy
	kennedys:JohnKennedyJr
	kennedys:PatrickBKennedy
kennedys:deathYear	1963
kennedys:firstName	John
kennedys:gender	kennedys:male
kennedys:lastName	Kennedy
kennedys:middleInitial	F
kennedys:middleName	Fitzgerald
kennedys:name	John Kennedy
kennedys:parent	kennedys:JosephKennedy
	kennedys:RoseFitzgerald
kennedys:photo	<http://www.topbraidcomposer.org/demos/kennedys/images/204px-John_F._Kennedy,_White_House_color_photo_portrait.jpg>
kennedys:profession	kennedys:congressman
	kennedys:navy
	kennedys:president
	kennedys:senator
kennedys:spouse	kennedys:JacquelineBouvier
kennedys:suffix	
rdf:type	kennedys:Person

Figure 2: Frame-based instance view in TopBraid Composer on the example of the Kennedys ontology

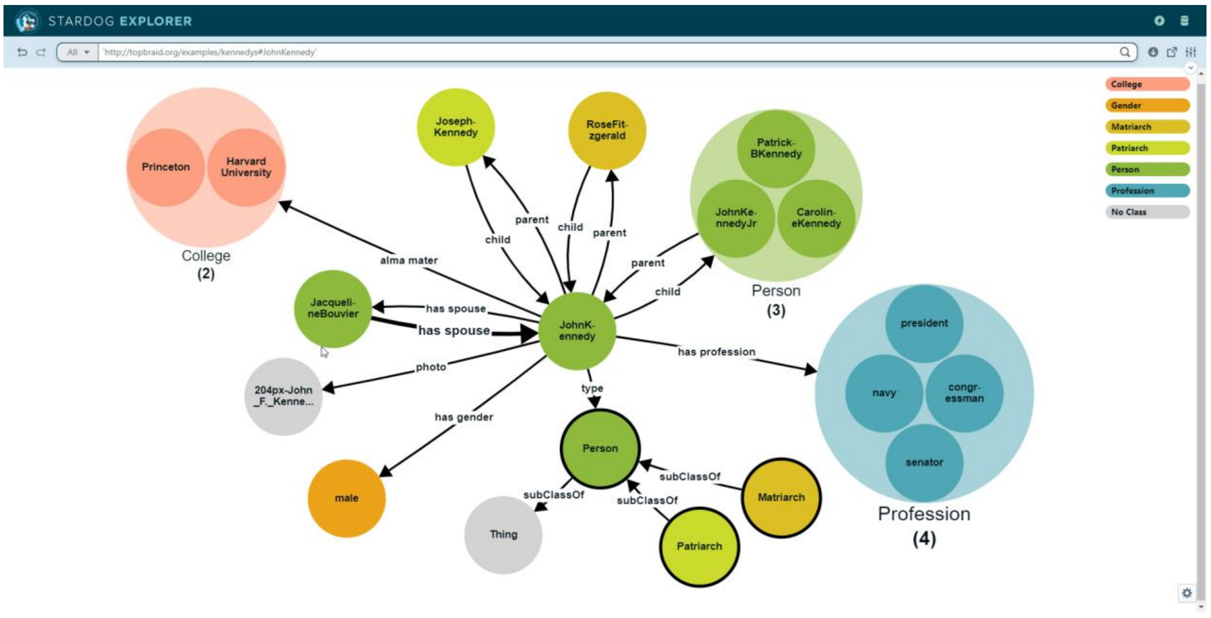


Figure 3: Graph visualization example in Stardog Explorer on the example of the Kennedys ontology

3. Virtual properties

Virtual properties are a concept that allows to display information of other parts of a KG directly for instances of a given class, as if it was a direct property of the instance. Virtual properties can help to improve the limitations of frame-based views (see Section 2) by adding additional distant information to the instance view. This makes the shown information richer and reliefs the need for navigating via paths in the graph to other instances to see their property values, in order to collect all the pieces of information required. Virtual properties cannot only improve the viewing of information of a KG, but also establish an improved instance search with advanced filtering capabilities.

The next sections explain virtual properties in detail, starting with a detailed concept, their syntactical definition in RDF, the computation of their values, and their utilization for an advanced instance search.

3.1. Concept of virtual properties

Virtual properties are a concept that allows to retrieve and show distant information, i.e. property values, connected with a given instance in the KG via a property path virtually. Virtual properties can be shown as if they were direct properties of an instance, together with the instance's other direct properties. They are "virtual" in the sense that these properties do not really exist in the KG, as OWL datatype and object properties do, but are instead automatically computed on demand at run time. As their values are virtually computed, they are read-only and cannot be changed by any user.

Virtual properties are defined as a path of properties. The path starts at a defined class (the "*domain class*"), traverses one or more object properties to neighbored nodes, and ends with the *final property*. If the final property is a datatype property, we speak of a *Virtual Datatype Property*. Otherwise, if it is an object property, it is a *Virtual Object Property*.

For better visualizing the concept of virtual properties, a KG is shown in Figure 4. The upper part shows an extract of the ontology TBox with some of its classes. The lower part comprises the instance graph (ABox), consisting of instances of the classes and their object property relations and datatype property values (literals). In this graph, the instance `d:ST350_CTR` of class `s:SystemController` has overall four "direct" properties: the datatype properties `rdfs:label`, `s:isReleased`, `s:typeNumber`, and the object property `s:hasSoftware`. All other information related to this system controller instance is defined in a subgraph that continues from the `s:SoftwareComponent` instance `d:SW_CTR_124` to the right hand side of the figure.

In order to show additional valuable information together with the instance `d:ST350_CTR` of class `s:SystemController`, four virtual properties have been introduced and are shown in the instance graph: the two virtual datatype properties `Software Release Date` and `Software Version`, and the two virtual object properties `Functions` and `Datatypes`. The virtual properties create shortcuts on the KG, as they turn multiple hops on the KG into a single hop that can appear as direct property of the domain class.

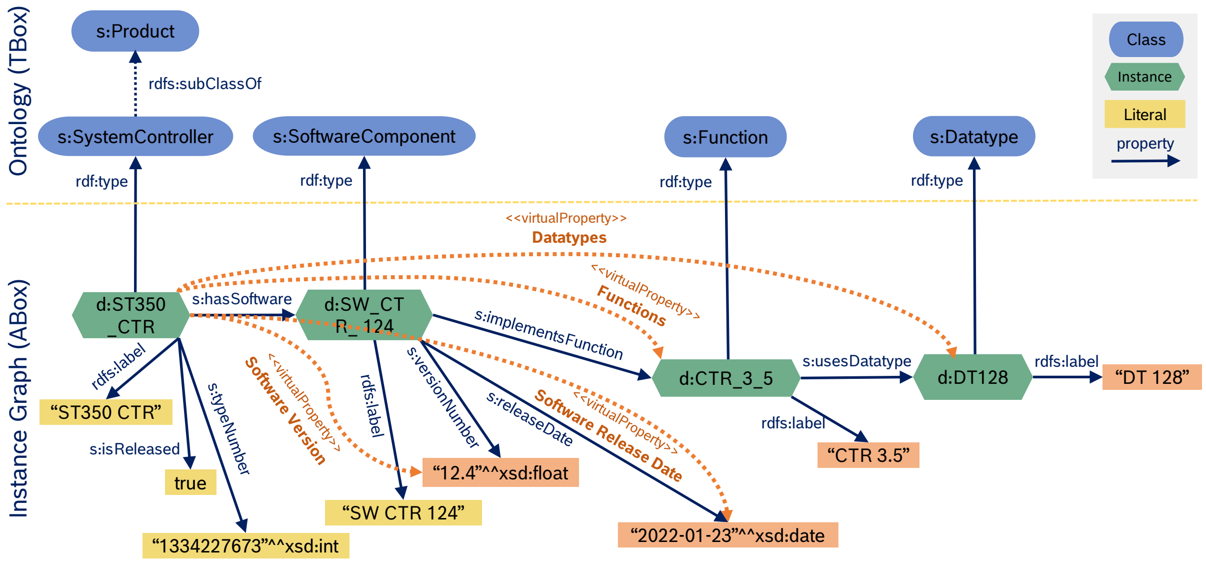
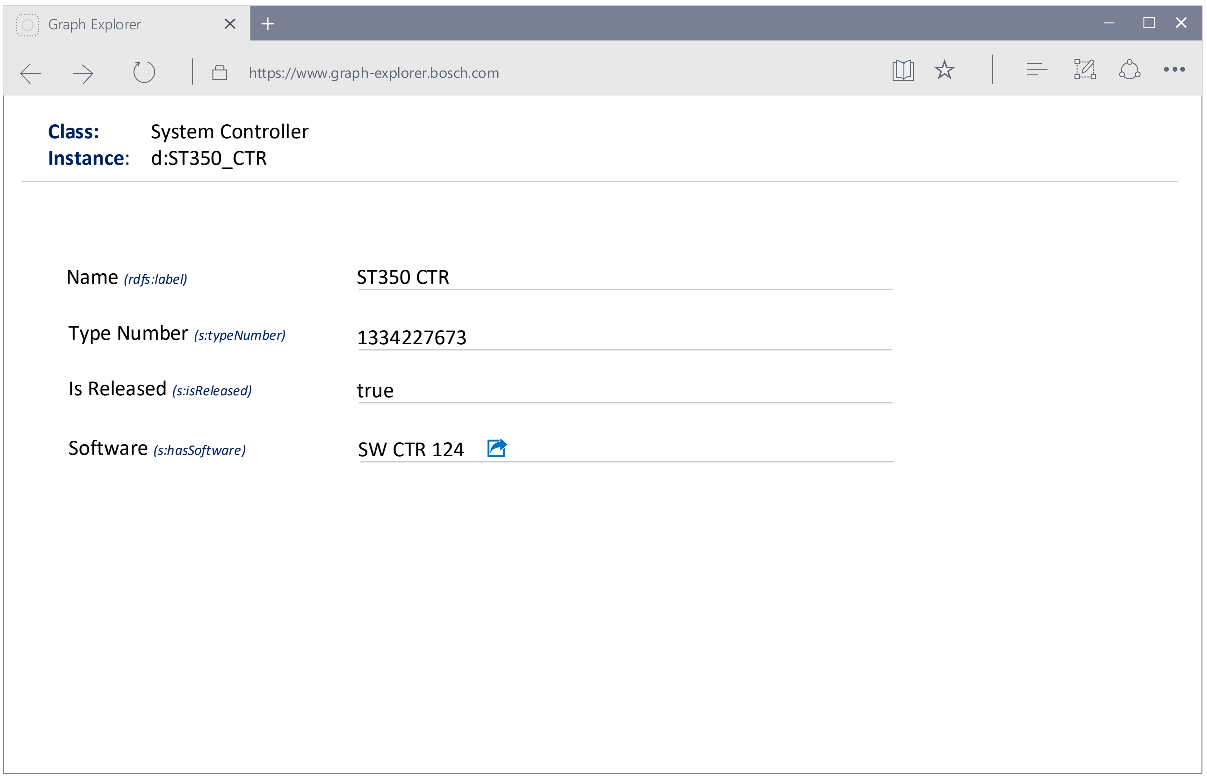


Figure 4: Knowledge Graph example of a system controller model with four virtual properties, highlighted in orange

When visualizing the instance `d:ST350_CTR` as frame-based view in a generic frontend, which fetches and shows the values of all properties (defined for the class) of the instance, it would normally be shown with its four direct properties only. This can be seen in Figure 5. Properties are therein shown with their name that is defined via the `rdfs:label` property in the ontology TBox, followed by their IRI in parentheses.

Enhanced by the concept of virtual properties as shown in Figure 6, the instance `d:ST350_CTR` has suddenly eight properties that appear as direct properties, but four of them are indeed the virtual properties. This view on the same instance provides much more details, which would normally only be accessible when further navigating through the graph along the connected instances `d:SW_CTR_124` of class `s:SoftwareComponent`, `d:CTR_3_5` of class `s:Function` and `d:DT128` of class `s:Datatype`. The virtual properties make this navigation and inspection of the other three instances (in many cases) unnecessary, and provide an enriched, condensed view on the instance `d:ST350_CTR` at a glance.

A specific mechanism that is applied to the frontend is to show the values of object properties, i.e. the instances in the object position of the property, not via their IRIs, but via their label property values. A *label property* can be `rdfs:label`, or any other string datatype property, if declared as label property in the KG. That is why the value of the "Software" property, for example, is not the instance IRI `d:SW_CTR_124`, but its `rdfs:label` value "`SW CTR 124`". The label property of a class can be defined via a specific annotation on the SHACL node shape of the class (TBox).

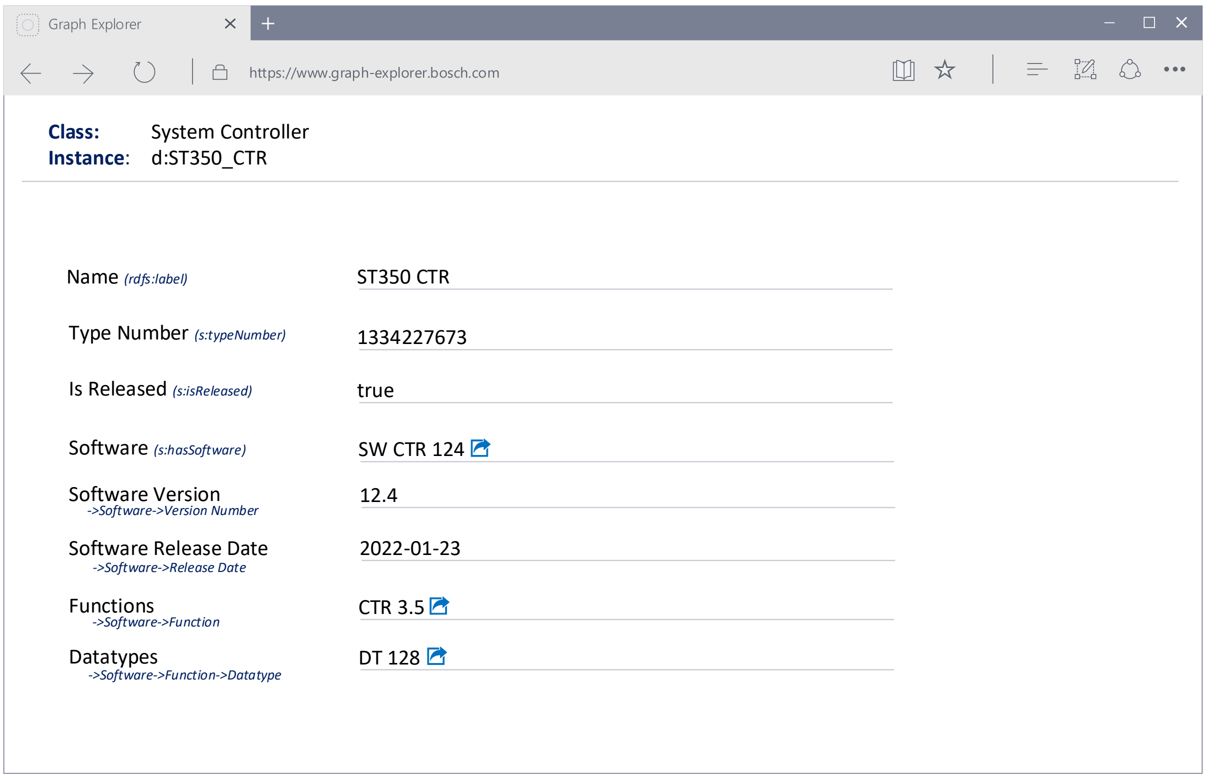

Graph Explorer

https://www.graph-explorer.bosch.com

Class: System Controller
Instance: d:ST350_CTR

Name *(rdfs:label)*	ST350 CTR
Type Number *(s:typeNumber)*	1334227673
Is Released *(s:isReleased)*	true
Software *(s:hasSoftware)*	SW CTR 124

Figure 5: Web-based UI concept for showing an instance in a frame-based view conventionally (direct properties only)


Graph Explorer

https://www.graph-explorer.bosch.com

Class: System Controller
Instance: d:ST350_CTR

Name *(rdfs:label)*	ST350 CTR
Type Number *(s:typeNumber)*	1334227673
Is Released *(s:isReleased)*	true
Software *(s:hasSoftware)*	SW CTR 124
Software Version *->Software->Version Number*	12.4
Software Release Date *->Software->Release Date*	2022-01-23
Functions *->Software->Function*	CTR 3.5
Datatypes *->Software->Function->Datatype*	DT 128

Figure 6: Web-based UI concept for showing an instance in a frame-based view enriched with four virtual properties

3.2. Definition of virtual properties

Virtual properties are defined by paths on a KG that they span. They turn these paths virtually into a direct property of the domain class, but they do not exist as materialized triples in the KG. Virtual datatype properties can appear like "normal", direct datatype properties in a UI, and virtual object properties like "normal" object properties. But virtual properties are not represented by explicitly defined OWL datatype or object properties. Instead, we decided to use SHACL as suitable language for defining virtual properties [8], a choice we also made for the definition of Property Assertion Constraints [9]. SHACL can define the expected structure of a KG via SHACL node and property shapes, as additional layer to the TBox. By the support of property chains, SHACL property shapes can also be repurposed for defining virtual properties. This approach is explained in the following using an example.

Figure 7 shows the definition of two virtual properties with SHACL - one virtual datatype property "Software Version" in the upper part, and a virtual object property "Datatypes" below. At the top of the RDF code, `s:SystemController` is defined therein as OWL class and SHACL node shape. The SHACL property `sh:property` is then used for attaching two SHACL property shape definitions to the SHACL node shape, one for each of the two virtual properties.

The SHACL property `sh:path` is typically used for defining the property of a property shape. But by its capability to also express SHACL property chains, it can be utilized for defining the paths of virtual properties also. For the virtual datatype property "Software Version", the path is defined as a chain of the object property `s:hasSoftware` and the datatype property `s:versionNumber` (see also Figure 4).

In addition to the standard SHACL properties (prefix "`sh`"), three new OWL annotation properties are introduced and used for defining virtual properties:

1. `vp:isVirtualProperty`: A boolean annotation property, used to explicitly declare a SHACL property shape to represent a virtual property definition.
2. `vp:pathLabel`: A string annotation property for attaching a human-readable path representation of the virtual property. This path label can be shown to a user in a frontend, such as in Figure 6. The value can be auto-generated from the concatenated `rdfs:label` values of all the properties in the path, separated via "->" strings, by using dedicated SPARQL/Update queries, for example. Whenever the labels of the properties exist in multiple languages, the path labels can be provided in these languages as language-tagged string values, with explicit language-tags, as shown in Figure 7.
3. `vp:sparqlPath`: A string annotation property for attaching the SPARQL property path representation of the SHACL property chain. This SPARQL path can be inserted into SPARQL queries to resolve the virtual property values by querying the KG. As the syntax of a SHACL property chain and a SPARQL property path are different (compare `sh:path` and `vp:sparqlPath` strings in Figure 7), a transformation from the one representation into the other is required. The transformation can be automated by using dedicated SPARQL/Update queries, for example.

Besides the three new annotation properties, a few other standard SHACL properties are used for completing the virtual property definitions: the SHACL property `sh:name` is used for defining a human-readable name of a virtual property, if required in multiple languages via language-tagged string values; `sh:nodeKind` distinguishes (non)virtual datatype properties (value "`sh:Literal`") from (non)virtual object properties (value "`sh:IRI`"); and `sh:datatype` is used for defining the datatype of virtual datatype properties, and `sh:class` defines the range class of virtual object properties.

It has to be noted that the only things to be defined by a user, e.g. an ontologist, are the `sh:path`, `vp:isVirtualProperty` and `sh:name` fields. All other fields can be auto-generated from the path (i.e. `vp:pathLabel`, `vp:sparqlPath`), and from the final property of the path, in particular its SHACL shape (i.e. `sh:nodeKind`, `sh:datatype`, `sh:class`). An option for the automatic generation, amongst others, are dedicated SPARQL/Update queries.

The property paths of virtual properties may also include and use inverse paths, meaning that the path points into the inverse direction of the object property. In SHACL this is denoted via `sh:inversePath` within `sh:path`, and in the corresponding SPARQL pattern via a leading "^" character in front of the object property IRI. Inverse paths allow to define virtual properties that navigate the KG along the incoming relationships of instances. Virtual property paths containing only inverse paths represent incoming relationships of length one or more. As an example of such an incoming virtual relationship, consider a virtual property starting from class `s:Datatype` in Figure 4, going backwards along the property `s:usesDatatype` to class `s:Function`, and further along `s:implementsFunction` back to class `s:SoftwareComponent`. This inverse virtual property would provide all the software components as values, in which a given datatype is used.

```
s:SystemController
  a owl:Class ;
  a sh:NodeShape ;
  sh:property [
    a sh:PropertyShape ;
    sh:path (
      s:hasSoftware
      s:versionNumber
    );
    vp:isVirtualProperty true ;
    vp:pathLabel "->Software->Version Number"@en ;
    vp:pathLabel "->Software->Versionsnummer"@de ;
    vp:sparqlPath "s:hasSoftware/s:versionNumber”
    sh:name "Software Version"@en ;
    sh:name "Softwareversion"@de ;
    sh:nodeKind sh:Literal ;
    sh:datatype xsd:float ;
  ] ;
  sh:property [
    a sh:PropertyShape ;
    sh:path (
      s:hasSoftware
      s:implementsFunction
      s:usesDatatype
    );
    vp:isVirtualProperty true ;
    vp:pathLabel "->Software->Function->Datatype"@en ;
    vp:pathLabel "->Software->Funktion->Datentyp"@de ;
    vp:sparqlPath
      "s:hasSoftware/s:implementsFunction/s:usesDatatype”
    sh:name "Datatypes"@en ;
    sh:name "Datentypen"@de ;
    sh:nodeKind sh:IRI ;
    sh:class s:Datatype ;
  ] ;
```

Figure 7: Definition of virtual properties as SHACL property shapes, using RDF turtle syntax

This shown approach of repurposing SHACL property shapes makes virtual properties ontology-defined entities inside the TBox. Such virtual property definitions state the type of property (`sh:nodeKind`), its name (`sh:name`), domain (the SHACL node shape to which the property shape is attached to) and range (`sh:datatype` / `sh:class`) explicitly, as it is the case for any other property. Virtual properties hence have everything that “normal” properties also have, which allows to include and visualize them in a similar way in a UI. Additionally, the path label can be shown in a UI. But as mentioned before, the values of virtual properties are to be computed virtually on demand, but not to be materialized (see next Section).

3.3. Computing the values of virtual properties

It has been mentioned already that virtual properties should not exist in a materialized form in a KG. This is for mainly two reasons: 1) it would be against the principle idea of being “virtual”; 2) it could cause inconsistencies when not properly kept in synch with updates of the KG. Instead, the values of virtual properties are to be computed on demand, whenever they are requested, e.g. by a UI that renders virtual properties.

There are at least two different options of how to compute the values of virtual properties: 1) via SPARQL queries; 2) via rules. OWL 2 property chains might also seem to be an option, but a detailed rationale of what speaks against them is discussed in Section 4.

The first option is to use the SPARQL path of virtual properties to generate dedicated SPARQL queries that can fetch the values. An exemplary SPARQL query pattern for that approach is shown in Figure 8. For any given instance and any of its virtual properties, the query can retrieve all values of the virtual property, by customizing it in the following way: the IRI of the instance needs to be inserted for the `%%instanceIRI%%` placeholder, and the SPARQL path of the virtual property (`vp:sparqlPath`) for the `%%sparqlPath%%` placeholder. The customization for the `s:System-Controller` instance `d:ST350_CTR` from Figure 4 and its virtual property “Datatypes”, for example, can be seen in Figure 9. For the given KG from Figure 4 the SPARQL query would return the instance `d:DT128` of class `s:Datatype`.

```
SELECT DISTINCT ?virtualPropertyValue
WHERE {
  BIND (%%instanceIRI%% AS ?instanceIRI).
  ?instanceIRI %%sparqlPath%% ?virtualPropertyValue.
}
```

Figure 8: General SPARQL query pattern for fetching the values of a virtual property for a given instance.

```
SELECT DISTINCT ?virtualPropertyValue
WHERE {
  BIND (d:ST350_CTR AS ?instanceIRI).
  ?instanceIRI s:hasSoftware/s:implementsFunction/
      s:usesDatatype ?virtualPropertyValue.
}
```

Figure 9: SPARQL query for fetching the values of the virtual property “Datatypes” of the instance `d:ST350_CTR` of class `s:SystemController`

For programmatically using this SPARQL-based approach, the functionality needs to be provided by an API. Such an API can be called by any client application that requires virtual property values, such as a UI (e.g. Figure 6). The API requires two input parameters from the client: the IRI of the instance, and the SPARQL path (`sh:sparqlPath`) of the virtual property. As its response, it provides a list of all values, to be further used (e.g. shown) by the client application.

The SPARQL query pattern from Figure 8 can of course easily be extended to comprise multiple virtual properties in a single query, so that multiple virtual properties of an instance can be retrieved at once. Likewise, the before mentioned API can be enhanced to return all property values of a given instance at once, including non-virtual ones. Such an enhanced API eliminates the need for repeatedly calling the API for each property individually.

The second option for computing the values of virtual properties are rules. Rule languages, such as SWRL [13] or Stardog rules [14], are suitable candidates. Both can infer the logical consequences of a rule on demand, when required. Stardog, for example, uses a query rewriting technique to include logical consequences of OWL or rules into the queries. When using rules instead of SPARQL, the SPARQL path (`vp:sparqlPath`) becomes obsolete. Instead of that, a rule needs to be generated for each virtual property from the SHACL paths. The equivalent SWRL rule for the SPARQL path "`s:hasSoftware/s:implementsFunction/s:usesDatatype`" from Figure 7 is displayed in Figure 10. Within this SWRL rule, the property IRI `s:virtPropDatatypes` is introduced artificially, without the need of defining it in the ontology TBox. When enabling the rules for the rule engine, the rule engine can take care of computing the values of virtual properties. Then, instead of using the SPARQL path of a virtual property, the virtual property IRI (e.g. `s:virtPropDatatypes`) can be used inside SPARQL queries directly. The utilization of the SPARQL paths is not required in this case for accessing the property values.

```
s:hasSoftware(?x1, ?x2) ∧ s:implementsFunction(?x2, ?x3) ∧
s:usesDatatype(?x3, ?x4) ⇒ s:virtPropDatatypes(?x1, ?x4)
```

Figure 10: SWRL rule for the virtual property "Datatypes" of class `s:SystemController`

3.4. Advanced Search Capabilities

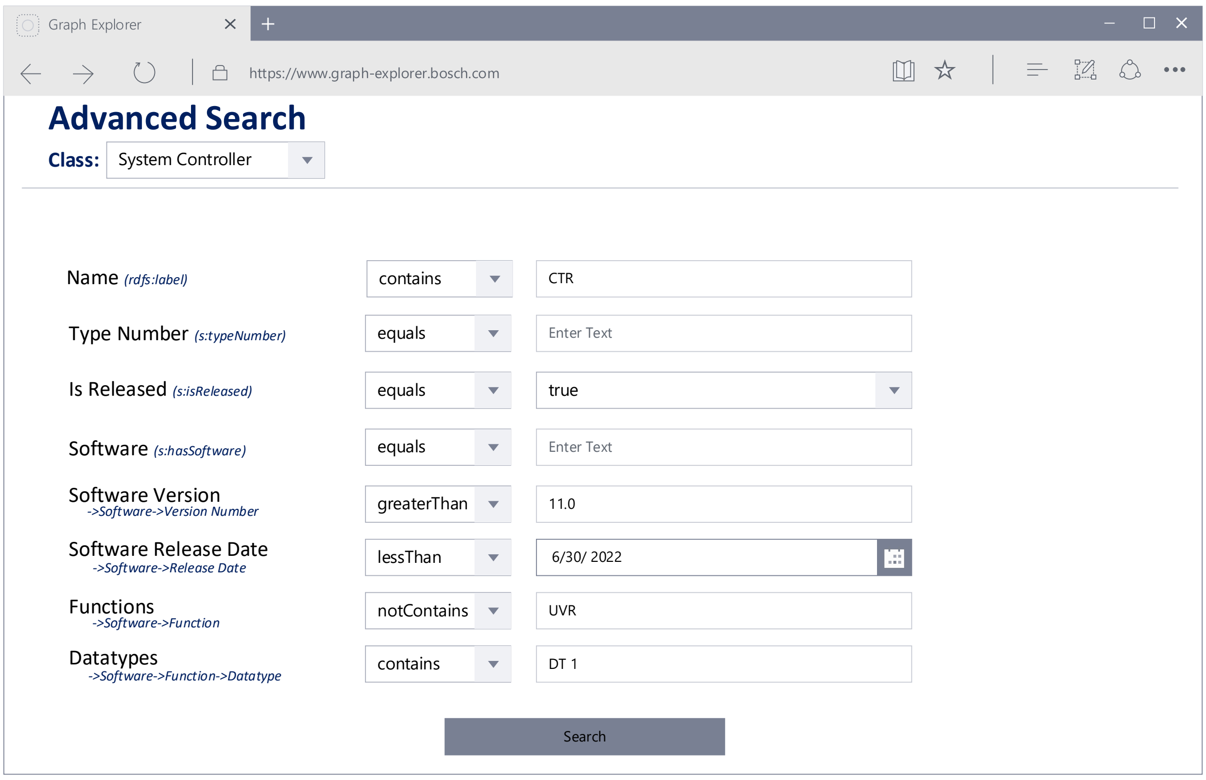
Besides showing additional information for a given instance, additionally to the direct properties of an instance, virtual properties can provide another important benefit, namely *advanced search capabilities*. When searching for instances of a given class satisfying certain criteria, then filters on the direct properties of the class can be defined quite easily. An example for the KG from Figure 4 could be a search for all `s:SystemController` instances, which have been released (datatype property `s:isReleased` and value `true`) and which have the substring "CTR" in their name (datatype property `rdfs:label`). This soley can be done using the direct properties of the class, in a simple UI, with a simple underlying SPARQL query.

But whenever information not directly accessible from an instance, but located at some other neighbored, neighbored, … instances is required within a search query, a bigger problem arises. Defining such search queries that could possibly span a large subgraph of a KG in a convenient and intuitive way via a user-friendly, fully generic UI is a very complicated problem, and good solutions seem to be still missing. Also, the query generation is much more complex then.

A suitable, user-friendly compromise for advanced filtering consists in the utilization of virtual properties, besides the direct properties. Generic search masks, which can be auto-generated for each class of a KG by using the TBox and SHACL shape definitions, can comprise both, the direct properties and virtual properties of a class. This is shown on the example of the system controller class in Figure 11. The UI shows an advanced search page, where the user can select a class of the KG via a dropdown. Immediately, a list of all direct and virtual properties of the class appear in the center of the UI. This search mask can be completely auto-generated from the KG on demand, and there is no need of implementing dedicated search masks for each class.

For each property inside the mask, the user can define a filter on the values of the property. For both, non-virtual and virtual properties, the same filtering options are provided, i.e. the characteristics and user experience is identical for the user. Depending on the type of property (for virtual properties, the type of the final property), different filter operators are available, such as:

- String datatype properties: `equals`, `notEquals`, `contains`, `notContains`
- Boolean datatype properties: `equals`
- Numerical (int, float etc.) and date/time properties: `equal`, `notEqual`, `lessThan`, `lessThanOrEqual`, `greaterThan`, `greaterThanOrEqual`
- Object properties: the filter operators of string datatype properties apply here, since the human-readable label values of instances are shown, and not the instance IRIs



Advanced Search

Class: System Controller

Name *(rdfs:label)*	contains	CTR
Type Number *(s:typeNumber)*	equals	Enter Text
Is Released *(s:isReleased)*	equals	true
Software *(s:hasSoftware)*	equals	Enter Text
Software Version *->Software->Version Number*	greaterThan	11.0
Software Release Date *->Software->Release Date*	lessThan	6/30/ 2022
Functions *->Software->Function*	notContains	UVR
Datatypes *->Software->Function->Datatype*	contains	DT 1

Search

Figure 11: Generic, auto-generated advanced search mask for the class system controller, offering all direct and virtual properties defined for the class

The advanced search UI needs to be comprehended by a suitable backend API, which takes the class IRI and the defined filters as input, and which generates a customized SPARQL query. Similar as the computation of virtual property values (see Section 3.3), the SPARQL paths of the virtual properties are inserted into the overall SPARQL query, comprehended by a corresponding `FILTER` statement, specific for the applied filter operator.

Such a SPARQL example query can be seen in Figure 12, showing a "`contains`" filter on the virtual property "Datatypes". This query returns all instances of class `s:SystemController`, which support a datatype (via a function implemented by its software) with the substring "DT 1" in their name. Of course, also multiple filters on different properties, e.g. the ones shown in Figure 11, can be combined into a single, complex SPARQL query, which returns all instances matching all the filters.

```
SELECT ?instanceIRI
WHERE {
  ?instanceIRI a s:SystemController.
  ?instanceIRI s:hasSoftware/s:implementsFunction/
      s:usesDatatype ?virtualPropertyValue.
  ?virtualPropertyValue rdfs:label ?label.
  FILTER (CONTAINS(?label, "DT 1"))
}
```

Figure 12: Principle of defining search filters on virtual properties, on the example of the virtual object property "Datatype" and filter operator "`contains`"

4. Differentiation from property chains in OWL, SHACL and SPARQL

The introduced virtual property approach is based on SHACL and applies *SHACL property chains* for defining the paths of virtual properties. The SHACL property `sh:path` is used for attaching the property chains to the property shapes, as can be seen in Figure 7. SHACL property chains are not meant for inferring facts, but they define the paths in a graph to which a SHACL constraint applies to.

Their original purpose is the validation of KGs. In the virtual property approach, however, they are used for defining the virtual paths in the KG. In order to compute them, the SHACL property chains are (automatically) transformed into *SPARQL property paths* (see Section 3.2). Two examples of such SPARQL property paths can be seen in Figure 7 as values of the property `vp:sparqlPath`. The SPARQL property paths are then inserted into a SPARQL query, which can compute the values of virtual properties dynamically on request (see Section 3.3).

It must be noted here that SHACL and SPARQL both support the definition of paths in a KG via SHACL property chains and SPARQL property paths respectively, but they are different languages with different syntaxes, and different objectives.

Yet another possibility to define paths in a KG are *OWL 2 property chains* [10]. The virtual object property “Datatypes” shown in Figure 7, for example, could be defined with OWL, instead of using SHACL. This can be seen in Figure 13, where the defined object property `s:supportedDatatypes` is the equivalent to the virtual property. Instead of SHACL properties, RDFS and OWL properties are used herein for defining the property’s domain (`rdfs:domain`), range (`rdfs:range`), (multilingual) label (`rdfs:label`) and path (`owl:propertyChainAxiom`). OWL 2 property chains can be interpreted by a reasoner, which can infer the object properties in a KG.

```
s:supportedDatatypes
  a owl:ObjectProperty ;
  rdfs:domain s:SystemController ;
  rdfs:range s:Datatype ;
  rdfs:label "Datatypes"@en ;
  rdfs:label "Datentypen"@de ;
  owl:propertyChainAxiom (
      s:hasSoftware
      s:implementsFunction
      s:usesDatatype
    ) ;
```

Figure 13: Alternative definition of the virtual property “Datatypes” as OWL 2 property chain

At a first glance one could argue that OWL 2 property chains already cover the functionality of virtual properties. But there are two fundamental reasons, why OWL 2 property chains are not a suitable solution. First of all, OWL 2 property chains per definition can only consist of object properties. This decision has been made for reasons of decidability, although some reasoners might support it beyond the OWL 2 standard. Virtual datatype properties however contain a datatype property as the last property in their path. OWL 2 property chains may hence only be used for virtual object properties, but not for virtual datatype properties. The virtual datatype properties “Software Version” and “Software Release Date” introduced earlier in this paper cannot be realized with OWL 2 property chains.

Secondly, it is a choice to make between the two fundamentally different but overlapping paradigms of OWL and SHACL. When using OWL, an application depends on description logic reasoning under the *open world assumption* (OWA), which basically means that it is not possible to assume that a statement is false, just because it is not declared in the ontology. Secondly, OWL does not adopt the *unique name assumption* (UNA), which means that individuals with different identifiers are in principle considered to be the same in OWL, unless explicitly stated differently. Reasoning under the OWA and UNA does not fit well to most of the real-world problems that we find in industrial use cases. Incomplete KGs cannot be properly detected under the OWA with OWL, more complex constraints cannot be expressed, and inferences are often not truly helpful, but rather misleading and disturbing [11].

Instead, many KGs nowadays use SHACL and not OWL for the definition of ontology axioms and constraints, e.g. the domains and ranges of properties (SHACL value type constraints); the expected minimum and maximum number of values per property (SHACL cardinality constraints); the lower and upper bounds of (comparable) literal values (SHACL value range constraints); the minimum and maximum length, the pattern and the language of string literal values (SHACL String-based constraints) and many more [8]. SHACL overcomes most of the complications and limitations of OWL and is becoming the industrial standard for KGs. It is well supported by the leading triple stores, allows for an extensive checking of completeness, is more expressive and supports far more complex constraints and rules than what can be formulated with OWL. Furthermore, SHACL purely relies on queries on a KG and does not require the computational overhead of a reasoning engine. Given all these points and the growing dominance of SHACL over OWL, the realization of virtual properties with SHACL, as introduced in this paper, makes most sense to us.

5. Evaluation

The virtual property approach has been implemented at Bosch and is being used currently by about 100 engineers in a productive setting at Bosch Thermotechnology for almost a year now. It is implemented as generic frame-based UI, which was first introduced in [12] for a semantic data lake. The frame-based UI is similar to the one shown in Figure 6 and Figure 11, and allows the users to view, browse and search the data in a KG, in a way as it has been explained in the previous sections. The virtual property approach is a central part of this overall architecture, that besides the browser-based UI also comprises a backend and KG storage layer. The KG is persistently stored in an RDF triple store. Currently, the solution supports the open-source triple store Apache Jena Fuseki [15] and the commercial triple store Stardog [16] as alternatives. In between the triple store and frame-based UI is a node.js based backend with dedicated REST APIs, which can fetch property values of instances (see Section 3.3) and which can realize the advanced search (both including virtual properties, see Section 3.4). Both types of APIs, and the browser-based UI have proven to be fast and responsive, even for large numbers of virtual or non-virtual properties and for large KGs. This has ensured a good user experience and user acceptance. From the productive usage we could gain many insights and positive feedback about virtual properties and the role that they play in improving the daily work of our engineers. In the following, these points will be briefly summarized.

The overall finding and feedback is that virtual properties can make a KG much better viewable, readable and searchable. Frame-based views are generally an intuitive, user-friendly way of showing information, and most of the users are used to them. Virtual properties can significantly improve the content and value of frame-based views on a KG, by enriching the scope of a class with other information beyond its direct neighborhood. These enhanced views on the KG allow users to more easily access, see and consume information, because they can see properties of other related instances in the graph within the same frame. It removes the necessity of navigating from one instance to others in many cases, which saves time and effort. In this way, virtual properties can considerably improve the limitations of conventional frame-based UIs.

Besides the improved readability, virtual properties can significantly improve the search functionality, without complicating the search definition. By additionally offering virtual properties in a search mask, besides the direct properties of a class, a search is no longer limited to the direct neighborhood, but can potentially span all parts of a KG, without the need of seeing or knowing the KG in its entirety. Even unexperienced users are able to use the advanced search functionality intuitively and define complex search queries far beyond the direct scope of a given class.

6. Conclusion

This paper presents virtual properties for ontologies and KGs. Virtual properties are virtually computed shortcuts on a KG connecting information at distant places of a KG, reachable via object property paths, directly to a given class. In a UI, virtual properties can appear just like direct, “normal” properties of a class, despite they do not exist in a materialized form.

An approach has been shown how SHACL property shapes can be enhanced and repurposed for defining virtual properties as ontology-defined entities. The on-demand computation of virtual property values is possible with SPARQL, or alternatively rules. A software architecture of a generic, self-adaptive, frame-based UI has been outlined, which can automatically generate views and masks for viewing and searching a KG, by just using the structural information defined in the TBox and SHACL shapes. The views and mask can automatically include virtual properties of the respective class, which has multiple benefits. Virtual properties can significantly improve the content and value of frame-based views, by enriching the shown information of a given instance with information from other, distant parts of a KG. Virtual properties can be included in the definition of search queries, which enables an advanced KG search far beyond the direct scope of a class.

Virtual properties support multilinguality, i.e. their names, paths and values (in case of string datatype properties) can be defined in multiple languages by using language-tagged string values.

The virtual property approach has been implemented at Bosch and is being used currently by about 100 engineers at Bosch Thermotechnology in a productive setting of a frame-based UI. This has successfully demonstrated the feasibility and significant value that virtual properties can provide.

7. References

[1] M. A. Musen, et. al. "The Protégé Project: A Look Back and a Look Forward." AI Matters 1.4 (2015): 4-12.

[2] M. Horridge, T. Tudorache, C. Nuylas, J. Vendetti, N. F. Noy, and M. A. Musen. "WebProtégé: a collaborative Web-based platform for editing biomedical ontologies." Bioinformatics 30.16 (2014): 2384-2385.

[3] TopQuadrant, TopBraid Composer. URL: https://archive.topquadrant.com/topbraid-composer-install/.

[4] TopQuadrant, Kennedys ontology. URL: https://www.topbraid.org/examples/kennedys.

[5] S. Lohmann, S. Negru, F. Haad, T. Ertl, User-Oriented Visualization of Ontologies, in: International Conference on Knowledge Engineering and Knowledge Management (EKAW 2014), Linköping, Sweden, 2014, pp. 266-281.

[6] Stardog Union, Stardog Exlorer. URL: https://docs.stardog.com/stardog-applications/explorer/.

[7] H. Dibowski, Ontology-Based Device Descriptions and Device Repository for Building Automation Devices, EURASIP Journal on Embedded Systems (2011) 1-17. doi: 10.1155/2011/623461.

[8] H. Knublauch, D. Kontokostas, Shapes Constraint Language (SHACL), W3C Recommendation, 2017. URL: https://www.w3.org/TR/shacl/.

[9] H. Dibowski, Property Assertion Constraints for ontologies and knowledge graphs, Data Technologies and Applications (2022) 1-20. doi: 10.1108/DTA-05-2022-0209

[10] J. Potoniec, "Inductive Learning of OWL 2 Property Chains." IEEE Access 10 (2022): 25327-25340. doi: 10.1109/ACCESS.2022.3155816

[11] H. Dibowski, K. Kabitzsch, Formal validation techniques for Ontology-based Device Descriptions, in: 16th IEEE Conference on Emerging Technologies & Factory Automation (ETFA), Toulouse, France, 2011. doi: 10.1109/ETFA.2011.6058974

[12] H. Dibowski, S. Schmid, Using Knowledge Graphs to Manage a Data Lake, in: R. H. Reussner, A. Koziolek, R. Heinrich (Eds.), INFORMATIK 2020, Gesellschaft für Informatik, Bonn, 2020, pp. 41-50. doi: 10.18420/inf2020_02.

[13] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission, 2004. URL: https://www.w3.org/Submission/SWRL/.

[14] Stardog Union, User-defined Rule Reasoning. URL: https://docs.stardog.com/archive/7.5.1/inference-engine/user-defined-rules.html.

[15] The Apache Software Foundation, Apache Jena Fuseki. URL: https://jena.apache.org/documentation/fuseki2/.

[16] Stardog Union, Stardog. URL: https://www.stardog.com/.