.15926 Editor

Version 1.4

Volume 2

APIs of Scanner and Builder

February 23, 2013

Volume 2. APIs: Scanner and Builder

Contents

Other documentation volumes:

Volume 1. Getting Started
Volume 3. Extensions
Volume 4. Patterns and Mapping

License

Parts of .15926 Editor (built-in extensions and extension samples) are released as a source code under the BSD 2-Clause license.

Other parts of the software (released in binary and in text form) are not covered by the license above and are distributed "as is" and free of charge for evaluation purposes only!

Elephant icon by Martin Berube is used for .15926 software according to terms at http://www.iconarchive.com/show/animal-icons-by-martin-berube/elephant-icon.html

8. SearchLan.15926

8.1. Query Functions

API functions of various .15926 Platform components are available in the Python console of the .15926 Editor and can be used in the Editor's extensions. For general information about Python console usage please refer to **Volume 1. Getting Started**, script examples can be found there also. Extensions of the Editor are described in **Volume 3. Extensions.**

API of Scanner component (called also **SearchLan.15926**) is described in this section of the documentation.

8.1.1. SearchLan general syntax

For all data sources containing a reference and project data (including template definitions), two query functions **find** and **show** are available in a Python console. Data source must be compliant to RDF/OWL serialization of ISO 15926 data supported by the .15926.

You can fully rely on the search results for reference and project data from local file data source only. Searches for reference and project data from SPARQL endpoint are limited to the data already downloaded for local storage; therefore results may be severely incomplete. Some capabilities to improve search at endpoints are described in the corresponding section.

If ***"Error: name '…' is not defined"*** error message is displayed in the console history panel – check whether indeed reference and project data source is open in your active panel or in a module you are addressing in your code.

General syntax for SearchLan functions is:

find(key=value)

show(key=value)

or

find(key1=value1, key2=value2, …, keyN=valueN)

show(key1=value1, key2=value2, …, keyN=valueN)

Function **show** also accepts a variable name:

show("var", key=value)

show ("var", key1=value1, key2=value2, …, keyN=valueN)

Some keys can be used with modifiers instead of values or in addition to values:

key=modifier

key=modifier==value

Double vertical quotation marks **"** are used as string delimiters in examples below. According to Python language rules it is possible to use double and single quotation marks interchangeably. Note that the construct **""** is two quotation marks in succession and signifies an empty string.

Unless stated otherwise all query examples below are run on local copy of PCA RDL – file ***RDL.owl*** downloadable from http://rds.posccaesar.org/downloads/PCA-RDL.owl.zip .

Some ISO 15926 data sources are really large and complicated queries can take a long time to complete. Use *File-Stop task (Ctrl+B)* menu command if you are waiting too long (direct SPARQL queries to an endpoint can not be interruptd by *Ctrl+B though*).

8.1.2. SearchLan functions details

Find

Function **find** searches the data source opened in the active panel for all entities matching listed conditions (where key equals value or belongs to value set). Found entities are returned to Python console environment as a set of strings – Python set object **set(["str1", "str2", ...])**. This set can be assigned to a named variable for further use, for example:

```
result = find(label=icontains("celsius"))
```

Strings in the returned set are either URIs of found entities (including type URIs if requested) or values of found annotation and literal properties, depending on the query target (see modifier **out** below). Function **find** is intended for use in nested queries in complex Python console scripts or in extensions.

Show

Function **show** returns the same results as **find** and also renders a search result for visual exploration in the active data source panel under the grouping node. The name of the node will be just *Found:* with a counter. The property grid for such node contains the full text of a query and allows copying.

If function is used with a variable name:

```
show("var", key=value, …, keyN=valueN)
```

then "*var"* is used to name search result node in the panel, and set of query results is assigned to the *var* variable for further use in console environment (only if *var* is a valid Python name, particulary if it doesn't contain spaces). If cursor focus is on the search result node – the grid shows the query used to form this node.

For example, the query:

```
show("res_cels", label=iendswith("celsius"))
```

returns all entities in the current graph whose label ends with "celsius" and shows them under a node named *res_cels,* at the same time a set variable *res_cels* becomes available for further work.

The query:

```
show(type=part2.Scale)
```

searches for all instances of Part 2 type Scale and places them under an unnamed node.

The query:

show("res_scales_cels", type=part2.Scale, label=iendswith("celsius"))

combines conditions from two previous examples to create a new search result node and a set variable *res_scales_cels*.

Function **show** should be used only at the top level of a query, call **find** function inside nested queries, or you'll get extra unnecessary nodes in the output.

Function **show** can be used only for data source open in active panel. Use function **find** with **with context(…)** wrapping to get data from one data source for use in searches in another data source.

Check

For nested queries an additional function **check** exists in SearchLan with the same syntax as **find**. If **find** is used inside another **find** or **show** – then innermost function is called first. If **check** is used instead of inner **find** – **check** is called after outer **find** or **show**. Search results returned will be the same, but execution speed can be optimized by careful use of **check**. Try **check** if you are restricting some annotation property by logical condition (see below for details).

For example, the first query:

show(type=part2.Classification, hasClassifier=find(label=icontains("UOM")))

will be executed by looking for all entities whose label contains substring "UOM" first, and then will look for all instances of Classification relationship where these entities are in *hasClassifier* role.

The second query

show(type=part2.Classification, hasClassifier=check(label=icontains("UOM")))

will start by looking for all instances of Classification in a data source, and then filter them by the label of the entity in *hasClassifier* role. Depending on the data source the first or the second query can be more optimal.

Function **check** can be combined with logical NOT operator, **show** and **find** do not allow this. Use **–check(condition)** in a query to find all entities for which **condition** is wrong.

For example, query:

show(type=-check(type=part2.ClassOfIndividual))

returns all entities which are not typed by Class type or any of its subtypes.

8.1.3. Special value **void**

Special value **void** can be used with any key to search for entities where this particular key is missing for an entity (whether it is type, role or annotation). Below you will find examples of **void** used with each appropriate key type.

8.1.4. Special value **any**

Special value **any** can be used with any key to search for entities where this particular key is present with any value, but is not **void**. For example, **show(id=any)** will show all entities which are subjects in the data source.

8.1.5. Special value **wrong**

Special value **wrong** can be used with an **id** key or with ***role*** keys to find entities which are not passing built-in verification tests of the Editor. More about verification tests can be found in **Volume 1. Getting Started**.

Verification of large datasets can take a significant time, so use other queries to restrict verified entities. Always use special key **collection** if you want to search for all suspicious entities directly in the results of other query. Query encapsulated by the **collection** key is executed first and narrows the search space before verification starts. For example, never use

~~**show(type=part2.ClassOfIndirectProperty, label=icontains('range'), id=wrong)**~~

use instead:

show(collection=find(type=part2.ClassOfIndirectProperty, label=icontains('range')), id=wrong)

Search for suspicious entities in role keys can be executed in two formats:

show(type=part2.Classification, hasClassifier='http://posccaesar.org/rdl/RDS4316259653', hasClassified=wrong)

show(collection=find(type=part2.Classification, hasClassifier='http://posccaesar.org/rdl/RDS4316259653'), hasClassified=wrong)

8.1.5. Set operations

Standard Python set operations can be used in the SearchLan query or executed in the console whenever the set objects are returned:

S1|S2 – set union;
S1&S2 – set intersection;
S1-S2 – set difference;
S1^S2 – set symmetric difference.

Set operation on URI sets always returns an URI set, probable one-element set or an empty set.

8.2. Keys and Values

Available keys (key kinds) for use in calls of **find, show** and **check** (with some restrictions) are:

- **type** – used to specify the types of entities or patterns to look for;
- **id** – used to restrict the set of URI;
- **collection** – used to do nested searches in some special cases (see below);
- ***role property*** *keys* **–** used to specify the role occupiers for roles in relationships, classes of relationships or template instances;
- ***annotation property*** *keys* - used to specify the values or value ranges for annotation properties;

- ***literal property** keys* – used to specify the values or value ranges for properties with XML schema typed values;
- ***object property** keys* – used to specify the role occupiers for custom object properties;
- ***universal** keys* – used to restrict all roles or all properties of required entities;
- ***pattern role** keys* – used to specify the patterns role occupiers.

8.2.1. Key **type**

Key **type** is used to specify the type for required entities. This key allows to restrict search either by specific entity types defined in ISO 15926-2 (Part 2) lifecycle integration schema, or by particular template. URIs of non-ISO 15926 entities can be used also, allowing search in other ontologies.

A value for **type** key can be specified directly as a single URI string or as an URI set object, obtained as a result of another query or through a set operation on other sets.

For example:

show(type="http://rds.posccaesar.org/2008/02/OWL/ISO-15926-2_2003#ArrangedIndividual", label=icontains("pump"))

show(type=set(["http://rds.posccaesar.org/2008/02/OWL/ISO-15926-2_2003#ArrangedIndividual", "http://rds.posccaesar.org/2008/02/OWL/ISO-15926-2_2003#AssemblyOfIndividual"]) , label=icontains("pump"))

For specific types (Part 2 or templates) **type** key value can be specified as a name with a module prefix, as described below.

If **type** key value in a query for template instances is specified as full URI of a template (or URI set), it is forbidden to use in this query any ***role*** or **object *property*** keys. There is no such restriction if **type** value is specified with module prefix.

Search for Part 2 type instances

For reference to Part 2 types type names with fixed module name prefix ***part2*** can be used as **type** key values. Type names are CamelCase type IDs as defined in Part 7. Such queries will work with data sources which use any one of three Part 2 namespaces known to the Editor:

PCA RDL
http://rds.posccaesar.org/2008/02/OWL/ISO-15926-2_2003# ;

RDS/WIP
http://dm.rdlfacade.org/data# ;

ISO 15926-8
http://standards.iso.org/iso/ts/15926/-8/ed-1/tech/reference-data/data-model# .

Examples of **type** queries:

show(type=part2.Specialization)
show(type=part2.ClassOfArrangementOfIndividual)
show(type=part2.Scale)

The Scanner expects that entity types are identified in a data source by *rdf:type* property with URI formed from CamelCase ID in one of three supported namespaces listed above. Particular namespace used in the data source should be registered as *Part 2 namespace* in the data source property grid (click at the data source name root node in a panel then find the field in the grid).

Search in PCA RDL requires that the namespace "*http://rds.posccaesar.org/2008/02/OWL/ISO-15926-2_2003#*" is registered as *Part 2 namespace* in the PCA RDL property grid (it is so by default).

If types of entities in a data source are not identified correctly, it will be necessary to look at the file content and try to determine the way Part 2 types are identified and assigned to entities, then edit this field in the grid.

Special type modifier **any** can be used to search for instances of some Part 2 type and all its subtypes:

show(type=part2.any.ClassOfArrangedIndividual)

will search for all instances of ClassOfArrangedIndividual or its subtypes.

The query:

show(type=part2.any.Thing)

will look for all typed entities in a data source file. If some or all entities in a data source have no names assigned as labels or annotations known to the Editor, name query in a search field at the top of a panel will not find them. In this case you can use URI search or use the console query above to identify some content in such data source.

Search for template instances

The simplest way to search for template instances is to use template URI as a value for **type** key as described above. However it is forbidden to use in such query any **role** or **object *property*** keys – the Editor will not be able to identify correctly the source for template definitions. There are no restrictions on property keys and some additional possibilities are open if corresponding template definition data source is present in the project and registered as a module before searching for instances of its templates.

Of one or more data sources (files or SPARQL endpoints) with template definitions are preset in the project, click on the project node in *Project* panel, look for *Modules* field in the property grid and double-click *Value* field. To add a new module name enter a new name in a key field (use only alphanumerical identifier starting with a letter and containing no spaces). The Editor will mark the entry as incomplete. Click on the drop-down menu and choose a data source. The Editor will mark as errors all non-unique module names, and will check whether the name is formed correctly! If any mistakes are found – The Editor will not allow you to save properties.

If standard *Proto and initial set templates file…* is opened through *File – Add file(s)…* menu, it is assigned module name *p7tpl* by default.

If definitions of some templates are available from SPARQL endpoint, you've to do the following to facilitate search for instances of these templates:

- add endpoint to the project;
- assign it a module name;

- open data source and run *All templates from data source* command from *Search* menu or from a Toolbar.

When module registration is completed, data source (local file) can be searched for instances of templates defined in this module. The search is done by using key-value pair **type=<*module_name*>.TemplateName**.

Add template set file ***p7tpl_enhanced.owl*** from folder *<samples>\tablan* (drop it one the Project panel). Assign it a *p7tpl* module name.

Add to the project ***Example_One.rdf*** file from the same folder. This file contains results of the TabLan to ISO 15926 model transformation; refer to **Volume 3. Extensions** for more details. Open ***Example_One.rdf*** data source in a panel with double click.

Run the following queries for ***Example_One.rdf***:

show(type=p7tpl.DescriptionByInformationObject)

show(type=p7tpl.ClassOfArrangementOfIndividual)

Special type modifier **any** can be used to search data source for instances of all templates defined in a particular module. Notice that syntax differs from that for the Part 2 types – templates do not form a hierarchical tree.

For example:

show("all_insts", type=p7tpl.any)

will find in the file ***Example_One.rdf*** all template instances of templates defined in *p7tpl* module.

Modifier **any** can be also used to search for instances of template and all templates specialized from it (recursively). For example,

show(type=iiptpl.any.ClassificationOfIndividual)

will find all instances of template **ClassificationOfIndividual** and all instances of its more then 40 specializations (this example is for template definitions in IIP sandbox, not for ***Example_One.rdf***).

Annotation property keys (see below) can be used together with **type** key with **any** modifier. The query:

show(type=p7tpl.any, annSource=icontains("data"))

will return from ***Example_One.rdf*** all instances of all templates having *annSource* property with substring "*data*" in annotation property value.

Use of role property keys (template roles with type or class restrictions) with modifier any **will deliver incomplete results for some template sets (those with separate URIs for parent and specialized template roles).**

File ***p7tpl_enhanced.owl*** with template definitions used in TabLan import contains a full set of p7tpl initial set templates and only one additional template. As URIs of common templates in two modules are the same, it is possible to add p7tpl initial set module to the Project panel (through

File – Add file(s)…- Proto and initial set templates file…) and run another query for the same data source ***Example_One.rdf***:

show("all_stand_insts", type=p7tpl.any)

This search will reveal all standard template instances, but miss all instances of one additional template. The difference can be shown by a query using set difference operation:

show(id=all_insts - all_stand_insts)

Search by class

In some data sources ***rdf:type*** may be used to record classification relationship from reference data classes or contain non-ISO 15926 entities. To search for arbitrary URIs in ***rdf:type*** property values you can use expressions with logical conditions for strings (listed in the section 8.2.4).

For example, query:

show(type=icontains("http://posccaesar.org/rdl/"))

will look for all entities in your data source which are classified by entities from PCA RDL namespace.

8.2.2. Keys **id** and **collection**

Key **id** is used to restrict the search by the URI set.

The set of values for **id** key can be specified directly as a single URI string or as an URI set, obtained as a result of another query, or through a set operation on other sets.

Direct specification of a single URI (return to PCA RDL file to run the following examples):

show(id="http://posccaesar.org/rdl/RDS1322684")

Direct specification of an URI set:

show(id=set(["http://posccaesar.org/rdl/RDS1322684", "http://posccaesar.org/rdl/RDS1322549"]))

Set operation can be used to unite and visualise results from two queries. One example was provided above at the end of previous section. Another query:

show(id=find(label=icontains("celsius")) | find(label=icontains("fahrenheit")))

will return all entities whose label contains substrings "*celsius*" or "*fahrenheit*".

Expressions with special logical conditions for strings (listed in the section 8.2.4) can be used as values for **id** key also, which may be useful in case of human-readable URIs. For example, the query like the following one:

show(id=icontains("non-person-interpretable_identifier"))

will be able to find URIs of the kind proposed in Part 6.

There is a special key **collection** which is essentially equivalent to an **id** key. This key allows combination of several searches by URI sets – search in the results of the other search. For example, there are two ways to search for entities with a label containing two substrings:

```
show(id=find(label=icontains("celsius")) & find(label=icontains("degree")))

show(id=find(label=icontains("celsius")),  collection=find(label=icontains("degree")))
```

The search defined in the **collection** key is always executed first. Therefore **collection** key should be always used for verification of some subset of data entities. It allows narrowing the set of verified entities before the start of verification. For example:

```
show(collection=find(type=part2.ClassOfIndirectProperty), id=wrong)
```

8.2.3. **Role property** keys

Role property keys are used in SearchLan to specify search restrictions on object roles of relational type instances – relationship, class of relationship or template instances.

Role property keys include all roles of relational Part 2 type instances (for example, *hasClassifier*, *hasApproved*, *hasWhole* or *hasEnd1Cardinality*).

Template role names are used as **role property** keys in searches for template instances if these roles are restricted by Part 2 types or by RDL items in template definitions (i.e. not by XSD schema types).

The set of values for a **role property** key can be specified directly as a single URI string or as an URI set, obtained as a result of another query, or through a set operation on other sets.

Search for instances of Part 2 type

Example of direct specification of **role property** key value for Part 2 type instances:

```
show(type=part2.Classification,
hasClassifier="http://posccaesar.org/rdl/RDS1322684")
```

returns all instances of Classification where class DEGREE CELSIUS is a classifier.

Query:

```
show(type=part2.Classification,
hasClassifier=set(["http://posccaesar.org/rdl/RDS1322684",
"http://posccaesar.org/rdl/RDS1322549"]))
```

returns all instances of Classification where classes DEGREE CELSIUS or DEGREE FAHRENHEIT are classifiers.

Query:

```
show(type=part2.Classification, hasClassifier=find(type=part2.Scale))
```

returns all instances of Classification where classifier is an instance of Scale.

Queries:

> **show(type=part2.Scale, hasDomain=void, hasCodomain=void)**
>
> **show(type=part2.ClassOfIdentification, hasPattern=void)**

return all instances with incomplete data model – with undefined mandatory roles. Open any entity in the results of these searches. You will see error sign on *Properties* group.

In the absence of **type** key **role property keys** are by default interpreted as roles of relational Part 2 types, therefore it is possible to omit **type** key and search by **role property** key only. Be aware that some relational types have unique role names, but some inherit role names from supertype.

For example:

> **show(type=part2.ClassOfIdentification, hasPattern=find(label=icontains("")))**

will return all instances of ClassOfIdentification with *hasPattern* role occupier having proper label.

The query:

> **show(hasPattern=find(label=icontains("")))**

will return all such instances of four relational types: ClassOfRepresentationOfThing and its three subtypes ClassOfIdentification, ClassOfDefinition and ClassOfDescription.

Search for template instances

If template instances are queried by role occupiers – template definition should be present in the project with module name assigned. Template definition module has to be accesses to find role URI by role name. Naming of template definition modules is described in *Search for template instances* section above.

Examples below are again given for the file ***Example_One.rdf*** included with the .15926 Editor:

> **show(type=p7tpl.ClassOfArrangementOfIndividual, hasClassOfPart=find(label=icontains("TR")))**
>
> **show(type=p7tpl.DescriptionByInformationObject, hasRepresented=find(label="System 1 Typical Design"))**

8.2.4. **Annotation** and **literal property** keys

Annotation search

Annotation property keys are used in SearchLan to specify annotation property values or value ranges. Both instances of Part 2 types and template instances can possess annotation properties.

Annotations available to Scanner are listed in the project property grid or in the data source property grid as *Annotations.* Each annotation property is registered there with a unique short name (used as a search key) and property URI can be viewed in the editing mode. If one short name is defined for two different properties in project and in data source – URI defined for the data source will be used.

Specific sets of annotation properties for PCA RDL (file and endpoint) or Part 4 OWL representation are built in the .15926 Editor and are added by default if these data sources are open through dedicated commands in *File* menu. Files imported from TabLan spreadsheet or from catalog JSON files also have their specific annotation sets. All other data sources are opened with default set of annotations which includes *rdfs:label, rdfs:comment* and set of annotation properties defined in the Part 8 (*annUniqueName, annTextDefinition, annSource, annNotes, annAdministrativeNote*, etc.).

Custom annotations used in a file are not automatically recognized at data source opening. If you know URI of such annotations and want to facilitate search for them – they can be added to a the source property grid or to the project property grid (if they are used in more then one data source). Double click the list in *Annotations*, put new annotations on blank lines as space-separated pairs **name URI**. If format is wrong – the Editor will indicate an error and will not allow to save the data. The Editor will not check whether URI is well formed or not!

For example: **extraProperty http://example.org/properties#extraProperty**

Added properties are saved in the project description file.

The search with only **annotation property key** will show template instances possessing referred annotations if corresponding template definitions module is not present in the project. Template instances will remain unrecognized objects (shown as not properly typed). Remove all template definition sources from Project panel and try the following query on the file ***Example_One.rdf***:

show(annSource=contains(""))

Literal property search

Literal property keys are used in SearchLan to specify values or value ranges of literal properties – properties restricted by XML schema types.

Some Part 2 type instances may possess literal properties (for example, Cardinality instances may have *hasMaximumCardinality* and *hasMinimumCardinality*, instances of six subtypes of ClassOfExpressInformationRepresentation may have *hasContent* property). Template instances can also possess roles with literal values, to be filled with numbers, strings, dates and times, etc. (for example, template p7tpl:BeginningOfTemporalPart has role *valStartTime* restricted by type *xsd:dateTime*, template p7tpl:CardinalityEnd1MinMax has roles *valMaximumCardinality* and *valMinimumCardinality* restricted by type *xsd:double*).

Literal properties available to Scanner are determined by Part 2 type system (available for review through *Data types* menu) or by template definitions. To perform literal property search for template instances template definition data source must be added to Project panel and registered as module as described in *Search for template instances* section above.

Annotation and literal property key values

Values for **annotation** and **literal property** keys are whole strings to match property values (case sensitive) or logical conditions with substring and inequality operators:

contains("*substr*"), **icontains**("*substr*") – property value contains "*substr*" substring, case sensitive for **contains**, case insensitive for **icontains**;

beginswith("*substr*"), **ibeginswith**("*substr*") – property value begins with "*substr*" substring, case sensitive for **beginswith**, case insensitive for **ibeginswith**;

endswith("*substr*"), **iendswith**("*substr*") – property value ends with "*substr*" substring, case sensitive for **endswith**, case insensitive for **iendswith**;

lt(*val*) – property value is less than *val*;

le(*val*) – property value is less then or equal to *val;*

gt(*val*) – property value is greater than *val*;

ge(*val*) – property value is greater than or equal to *val*.

For inequality condition Python attempts to convert values of the property searched to the type represented in condition.

If *val* is an integer or floating point number, an attempt is made to interpret values of property searched as numbers, and Scanner will return an error in console history panel if it can not perform such conversion. For example, looking for a textual annotation property with condition **gt**(2) will result in an error if at least one property value has non-digital symbols.

If *val* is a string in **""**, then values of a restricted property are interpreted as strings and inequality is assessed using lexicographical order. Thus it is always possible to restrict a textual property with condition **gt**("2"), but remember that "11"<"2" despite the fact that 11>2.

Lexicographical date comparison also has to be performed carefully, as "2010.12.31"<"2011"<"2011.01.01".

Examples of annotation property queries (for PCA RDL file):

show(label="CUBIC METRE PER NORMAL CUBIC METRE, 0 DEGREE CELSIUS")

show(label=iendswith("Celsius"))

show(label=ibeginswith("degree"))

show(hasStatus=icontains("incomplete"))

show(hasNoteAdmin=icontains("incomplete"))

show(type=part2.Class, hasCreator=icontains("mvs"))

Query for missing annotation property:

show(type=part2.ClassOfScale, hasRegistrarAuth=void)

returns all instances of ClassOfScale where registration metadata is incomplete – information about registration authority is missing.

Query for entities with missing labels:

show(type=part2.ClassOfClass, label=void)

Query for missing literal property:

show(type=part2.Cardinality, hasMaximumCardinality=void)

Examples of numeric literal property queries for template instances in the file ***Example_One.rdf***:

show(type=p7tpl.CardinalityEnd1MinMax, valMinimumCardinality="1")

show(type=p7tpl.CardinalityEnd1MinMax, valMinimumCardinality=ge("0"))

show(type=p7tpl.CardinalityEnd1MinMax, valMinimumCardinality=ge(0))

show(type=p7tpl.CardinalityEnd1MinMax, valMinimumCardinality=ge(0.0))

Notice that for the exact role match only string type value can be used, but for the role typed ***xsd:double*** there are three equivalent ways to write inequality condition.

Value conditions for annotation and literal property key can be combined with logical operators:

-condition – logical NOT;
condition1&condition2 – logical AND;
condition1|condition2 – logical OR.

Example for PCA RDL:

show(label=icontains("degree")&-icontains("celsius"))

8.2.5. **Custom object property** keys

Custom object property keys are used in SearchLan to specify the role occupiers for those object properties which are used to record relationships in non-reified way, not recommended for ISO 15926-8 RDF representation.

Unlike **role property** keys, **custom object properties** should be registered as *Roles* in the properties of a project or of a data source before further use. Each **custom object property** is registered there with a unique short name (used as a search key) and property URI can be viewed in the editing mode. If one short name is defined for two different properties in project and in data source – URI defined for the data source will be used.

Short names registered in the property grid are used as **custom object property** keys.

All data sources in the Editor have one **custom object property** registered by default. It is

subClassOf http://www.w3.org/2000/01/rdf-schema#subClassOf

The set of values for a **custom object property** key can be specified directly as a single URI string or as an URI set, obtained as a result of another query, or through a set operation on other sets.

To see an example of **custom object property** search add reference and project data file ***sample_lookup.rdf*** from <*samples*>\ folder.

Do the query:

show(type=part2.any.Thing, subClassOf='http://posccaesar.org/rdl/RDS327239')

The single entity which is registered as subclass of PUMP (from PCA RDL) will be returned.

Or you can search file ***xyz-corp-rdl-extension.owl*** located in *<samples>\pid* with registered custom object property **subClassOf**:

> **show(type=part2.any.Thing, subClassOf='http://www.rdl.xyz-corp.com/data#Cc40374c0-e864-11e1-aff1-0800200c9a66')**

8.2.6. **Universal** keys

Key **object**

Universal key **object** is used to define a restriction for all object properties, without indication which particular property you are looking for. The value is checked against all type declarations, all roles of relational Part 2 type instances and all template roles for template instances, all registered custom object properties. Key **object** can be combined with other keys.

The set of values for an **object property** key can be specified directly as a single URI string or as an URI set, obtained as a result of another query (find or check), or through a set operation on other sets.

For example, the query:

> **show(type=part2.any.ClassOfRelationship, object="http://posccaesar.org/rdl/RDS327239")**

returns from PCA RDL all instances of ClassOfRelationship type and all its subtypes where class PUMP occupies one of the roles.

The query

> **show(type=part2.any.ClassOfCompositionOfIndividual, object=find(label=icontains("motor")))**

returns all classes of composition where either class of part or class of whole role is occupied by some entity with "*motor*" in name.

The query:

> **show(object="http://posccaesar.org/rdl/RDS8645837")**

returns all relational entities where ISO 15926-4 POSSIBLE INDIVIDUAL plays some role.

Universal key **object** can be used for template instance search in the file ***Example_One.rdf***:

> **show(type=p7tpl.any, object=find(label="TR-123.2"))**

returns all template instances (of any template used) where some role is occupied by an entity with the label "*TR-123.2*".

You can again search file ***xyz-corp-rdl-extension.owl*** located in *<samples>\pid* with registered custom object property **subClassOf**:

> **show(object='http://www.rdl.xyz-corp.com/data#Cc40374c0-e864-11e1-aff1-0800200c9a66')**

It is possible to use Part 2 types as values for **object property** key, if in some non-standard representation these types are occupying roles in some registered custom object properties. If namespace for Part 2 types is known, use the query with full URI:

show(object='http://rds.posccaesar.org/2008/02/OWL/ISO-15926-2_2003#ClassOfInanimatePhysicalObject')

If you want to look for types in any of three predefined Part 2 namespace, use query modelled like:

show(object=part2.ClassOfInanimatePhysicalObject.uri)

As ***rdf:type*** is an object property, for PCA RDL file these two queries will return correspondingly typed entities.

Key **literal**

Universal key **literal** is used to define restriction for some literal or annotation property of an entity, without knowledge of the specific property name or URI. It is applied to all annotation properties registered for entities in a data source and to all literal properties defined by Part 2 or by template definition. Key **literal** can be combined with other keys.

Values for **literal property** key are whole strings to match property values (case sensitive) or special logical conditions listed in the section 8.2.4 above. For a successful match some property value has to satisfy the whole condition.

Be aware that numerical conditions with integer or floating point numbers are practically useless here – the Scanner will try to transform values of all literal properties met in the data source to numeric format and almost certainly will report an error on some text or date string.

The following query for PCA RDL:

show(literal=icontains("air"))

returns all entities where substring "*air*" is used in definitions, labels, creators names, other string values of annotations or literal properties.

The query :

show(literal=icontains("motor")&icontains("electric")&icontains("class"))

returns those entities which have in some (one!) property value all text fragments listed in the query.

If you want to construct a query for an entity which has all text fragments but probable located in different literals properties – you should search for all required substrings sequentially with nested **find** function calls.

For example, modify the previous query to:

show(literal=icontains("motor"), id = find(literal = icontains("electric"), id = find(literal = icontains("class"))))

and see extra result compared to previous query.

Key **name**

Universal key **name** is used to define restriction on a predefined subset of annotation and literal properties – on properties usually used for human-readable naming of entities. These are the same properties which are queried by a name search through a *search* box in data panels.

Key **name** should not be used in pattern search!

Properties are searched in the following order:

1. *rdfs:label*
2. pca:hasDesignation
3. dm:hasContent
4. meta:annUniqueName

If the property from this list is present for an entity and doesn't match search restriction, subsequent properties for this entity are not checked. If property is absent, the search checks for the next property. The search returns an entity as soon as some property matches the restriction.

Values for **name** key are whole strings to match property values (case sensitive) or special logical conditions listed in the section 8.2.4 above. It makes this search more powerful compared to the name search through *search* box at the top of a data panel. *Search* box performs a case-insensitive search for simple substring, while console search allows a case-sensitive search, search for beginnings and ends of a name, and logical combination of conditions.

Key **name** can be combined with other keys.

8.2.7. Modifiers

Modifier **out**

Modifier **out** is used on a key if we are searching for a particular property values for entities found by an unmodified query. Modifier **out** can be used for property keys in both **find** and **show** SearchLan functions.

Modifier **out** will be ignored if it is used on a key inside a **check** function.

For example, query:

```
show(type=part2.Classification, hasClassified=out,
hasClassifier="http://posccaesar.org/rdl/RDS1322684")
```

returns set of entities classified by DEGREE CELSIUS, not set of Classification instances as unmodified query should.

Query:

```
show(type=part2.PropertyQuantification, hasInput=out,
hasResult=find(type=part2.RealNumber))
```

returns all properties in RDL which are quantified by instances of RealNumber.

Modifier **out** can be used simultaneously with value restriction of a same key. For example, query:

show(hasClassified=out==check(label=icontains("bar")), hasClassifier="http://posccaesar.org/rdl/RDS4316259687")

returns all Scales classified by ISO TS 15926-4 (2007) UOM CLASS whose label contains substring "*bar*".

If modifier **out** is used on a literal, it adds additional condition "exists" on its value. For example query:

show(hasCreator=out)

looks like a query without any restriction, but it really returns the list of creators registered for all reference data entities in PCA RDL (without further links to entities created!).

Some data sources compliant to Part 8 can use annotation property *meta:annUniqueName* instead of *rdfs:label* to store human readable identifications. Try the console query:

show(annUniqueName=out)

It will deliver the list of unique names for all named entities in the file (without links to their URIs!). This query differs from search for **type=part2.any.Thing** which returns named and unnamed entities together.

Query:

show(type=out==part2.any.Thing)

returns all Part 2 types which are instantiated in the data source.

If some template definition data source is registered with module name **tpl**, then query:

show(type=out==tpl.any)

returns all templates from **tpl** module which are instantiated in queried project data source.

Query:

show(type=out)

returns all entities which are used as types for entities in a data source, as Part 2 types, templates or RDL superclasses, OWL types.

Avoid using **out** more then once on the same level of a query – the software will process only one modifier, in unpredictable manner.

Modifier **groupby**

Modifier **groupby** is used for grouping search results in subgroups by values of key marked with **groupby**. Modifier **groupby** can be used on keys only in **show** function. Notice that **id** key doesn't allow use of **groupby**.

If modifier **groupby** is used, the counter in the *Found* node group title shows the number of subgroups formed, not the number of entities found.

For example, query:

show(type=part2.Scale, label=icontains("celsius"), hasDomain=groupby)

returns all instances of Scale having "*celsius*" in label grouped by distinct domains (property spaces).

Compare to query:

show(label=icontains("celsius"), hasDomain=groupby)

which returns in addition to results of previous query instances of types other then Scale having "*celsius*" in label. These other type instances do not have hasDomain property; therefore additional search results are present on top level outside of any group.

Query:

show(type=part2.ClassOfClass, hasCreator=groupby)

returns all instances of ClassOfClass grouped by registered creator.

Modifier **groupby** can be used simultaneously with value restriction of a same key. For example, query:

show(type=part2.Scale, label=icontains("celsius"), hasDomain=groupby==check(label=icontains("capacity")))

returns those instances of Scale which have "*celsius*" in label and simultaneously have domain in property spaces with "*capacity*" in label. The results are returned grouped by domains.

Next query:

show(type=part2.Classification,hasClassified=out==check(type=part2.PropertyQuantification), hasClassifier=groupby==check(type=part2.Scale))

will return all property quantification relationships grouped by instances of Scale that classify them.

Another example:

show(type=part2.ClassOfClassOfComposition, hasClassOfClassOfPart=out, hasClassOfClassOfWhole=groupby==check(type=part2.DocumentDefinition))

shows whole-part relationships in the domain of document models.

If it is required to use modifier **groupby** and simultaneously restrict a key by elements of a known set of URIs, the set value can be assigned to the role key:

key=groupby==set(["URI1", "URI2", …])

For example, the query:

show(type=part2.Classification, hasClassifier=groupby==set(["http://posccaesar.org/rdl/RDS1322684", "http://posccaesar.org/rdl/RDS1322549"]))

returns all instances of Classification where classifier is either DEGREE CELSIUS or DEGREE FAHRENHEIT scale, grouped by these classifiers.

If grouped search results are assigned to a variable by use of variable parameter in **show** function, resulting set consists of URIs pairs – search result URI and grouping node URI – separated with vertical bar "|".

For example, the query

show("res_list", type=part2.Scale, label=icontains("celsius"), hasDomain=groupby)

assigns to the variable **res_list** the following set of URI pairs (only part of the set shown):

```
set([
'http://posccaesar.org/rdl/RDS1322684|http://posccaesar.org/rdl/RDS355859',
'http://posccaesar.org/rdl/RDS14122227|http://posccaesar.org/rdl/RDS14119855',
'http://posccaesar.org/rdl/RDS14152900|http://posccaesar.org/rdl/RDS382049',
...])
```

where in each pair URI of a Scale instance is before the "**|**" sign and URI of an entity in **hasDomain** role is after it.

Avoid using **groupby** more then once on the same level of a query – the software will process only one modifier, in unpredictable manner.

8.3. Pattern search

About patterns

Pattern recognition in data sources is a powerful mechanism of .15926 Platform. Patterns libraries for the Editor are located in Python files with ***.py*** extension, placed in the *<installation_folder>/patterns* folder.

Use of patterns in search is described in this section. Released version of .15926 Editor has an initial set of predefined patterns defined in the library ***pattern_samples.py.*** It depends on three template data sources:

- Part 8 initial set in a module named ***p7tpl*** (the file ***p7tpl.owl*** accompanying ISO 15926-8 is included with the distribution in folder <*samples*>);
- PCA MMT SIG set in a module named ***projtpl*** (the work-in-progress version of PCA MMT SIG set is included with the distribution as ***templates.owl*** file in folder <*samples*>*\pid*);
- IIP template set in a module named ***iiptpl*** available as **IIT Sandbox (templates)** from *Files* menu.

Users can get more pattern libraries as we release them, expand existing patterns and add new patterns. Please refer to **Volume 4. Patterns and Mapping** for more details.

Initial set of patterns is built on the basis of RDF predicates, Part 2 types, Part 8 initial template set and PCA MMT SIG proposed template set (the set currently discussed by the Special Interest Group Modelling, Methods and Technology of POSC Caesar Association and available at http://15926.org). Use of pattern definitions depends on template definitions being added to the project and assigned module names specified in pattern definitions. An absence of a registered template definition module will lead to inability to recognize some patterns.

Remember that data panel pattern visualization options set in the project properties (as described in **Volume 1. Getting Started**) do not influence your ability to search for all patterns from all pattern libraries present.

For use with the initial patterns Part 8 template set module name should have a name ***p7tpl***, and PCA MMT SIG template set module name should be ***mmttpl***

Part 8 initial set is included with the distribution as ***p7tpl.owl*** in folder <*samples*> and should be opened through menu *File – Add file(s)…- Proto and initial set templates file…*The version of PCA MMT SIG set (with some namespace changes for compatibility) is included with the distribution as ***templates.owl file*** in folder <*samples*>*\pid*.

Pattern definitions can be changed without closing of the Editor and reloaded with *File - Reload patterns* menu command ***(Ctrl+Shift+W).*** New patterns will appear in *Patterns* node groups of your data sources after the reload, but you should reload (*F5*) those entities where *Patterns* node group was already expanded!

<u>*Pattern search keys*</u>

In pattern search pattern names and pattern option names are used as **type** key values, and pattern role names are used as ***pattern role*** keys to restrict your search.

Available pattern names can be found by ***patterns*** command in console (make sure that reference and project data source is open in active panel).

Pattern role names and option names for a pattern can be found by ***patterns.<PatternName>*** command in console. If some options are listed as *Unnamed_N* – pattern search can not be restricted to these options.

General syntax for pattern search is:

show(type = patterns.*PatternName*, patternRole1=value1, …, patternRoleN=valueN)

find(type = patterns.*PatternName*, patternRole1=value1, …, patternRoleN=valueN)

If only ***PatternName*** is specified, the search will look for all options in the pattern. To restrict search by only one option – specify also an ***OptionName***:

show(type = patterns.*PatternName.OptionName*, patternRole1=value1, …, patternRoleN=valueN)

find(type = patterns.*PatternName.OptionName*, patternRole1=value1, …, patternRoleN=valueN)

If one of the options in a pattern is identified as an expansion of the other option, expansion option can be accessed by using **.expansion** extension:

show(type = patterns.*PatternName.OptionName*.expansion, patternRole1=value1, …, patternRoleN=valueN)

find(type = patterns.*PatternName.OptionName*.expansion, patternRole1=value1, …, patternRoleN=valueN)

The set of values for a **pattern role** key can be specified directly as a single URI string or as an URI set, obtained as a result of another **find** or **check** query, or through a set operation on other sets.

Pattern search output

Modifier **out** can be used on **pattern role** key. Modifier **groupby** is not allowed for **pattern role** keys.

If **out** is used on a **pattern role** key, results of **show** query are visualized as group of found entities and SearchLan function call returns set of found URIs as usual.

For example, query for PCA RDL:

```
show(type=patterns.Composition, whole="http://posccaesar.org/rdl/RDS415124",
part=out)
```

returns all classes of parts for class ELECTRIC MOTOR.

And query:

```
show(type=patterns.Composition, whole="http://posccaesar.org/rdl/RDS364686618",
part=out)
```

returns the only individual impeller which is part of the individual PUMP P-101.

These two queries show how the same Composition pattern is defined for ontologically different entities – a class and an individual.

If **out** is not used in pattern search, results of **show** query are visualized as a special tree of pattern signature groups. For example, execute the query:

```
show(type=patterns.Specialization, superclass=find(label=icontains("uom class")))
```

and look at the results. You will see all superclass-subclass pairs where superclass label containing substring *"uom class"*.

Another example, using **.expansion** extension to look for patterns corresponding to template axiom:

```
show(type=patterns.PropertyOfClass.PropertRangeRestrictionOfClassTemplate.expa
nsion)
```

If **out** is not used in pattern search, results of a SearchLan function call are assigned to a variable in special *signature dictionary* format:

```
{
        'patternRole1': ['URI11', 'URI12', 'URI13', ...]
        'patternRole2': ['URI21', 'URI22', 'URI23', ...]
        'patternRole3': ['URI31', 'URI32', 'URI33', ...]
        ...
}
```

In this dictionary pattern role names are used as keys, with lists of found URIs as values. All lists have the same size. If some result is returned with incomplete set of roles (that may be allowed by some pattern options), special value **'None'** will be placed in missing role's list at the corresponding position.

To obtain for further processing all patterns for an entity or for a group of entities – use special function call:

```
patterns.get(URI)
patterns.get(URISet)
```

This function returns a dictionary of dictionaries, where top-level keys are pattern names and their values are *signature dictionaries* in a format described above.

For example, execute in the console:

```
pat_em=patterns.get("http://posccaesar.org/rdl/RDS415124")
print(pat_em)
```

and look at the output in the history area.

Universal keys in pattern search

Universal keys **object** and **literal** can be used in pattern search. Key **name** should not be used in pattern search.

Universal key **object** defines a restriction for all ***pattern role*** keys with object value, without indication which particular key should be restricted. For example, query:

```
show(type=patterns.Composition, object="http://posccaesar.org/rdl/RDS415124")
```

returns all composition pairs where class ELECTRIC MOTOR is part or whole.

Universal key **literal** defines a restriction for all ***pattern role*** keys with literal value, without indication which particular key should be restricted. For example, query:

```
show(type=patterns.Identification, literal=icontains("pump"))
```

returns all identification pairs where literal role contains substring "pump".

Notice that it is possible for a pattern role to have an object value in one pattern options and a literal value in another pattern options. For examples refer to predefined Identification pattern description (***patterns_samples.py*** file in ***<installation_folder>/patterns*** folder).

8.4. Search in template definitions

Template definitions are processed like regular reference data items starting from version 1.3 of the Editor, and SearchLan querying capabilities can be fully used for them.

It is possible to search for all types of annotations which can be used in template definitions or in template role declarations. Template specializations are identifiable through search for **patterns.Specialization**.

However there are many important peculiarities of template definition representation. See below for some instruments and tips for template definition searches.

Experienced Python programmer can extract full data on any template signature from dictionary ***<module_name>.<TemplateName>.template*** and use it in an adapter or in a verification extension.

Identifying templates

Template definitions are complex RDF subgraphs, and their identification in the data source is not an easy task. To help in this operation special construct is added to the Scanner API for use from the Python Console or in extensionы: **<*module_name*>.any** .

Its use as **type** key value for template instance search is described above. To search for template definitions, use it as **id** key value. For example:

```
show(id=tpl.any)
```

returns all template definitions from a data source with **tpl** module name.

It can be combined with other literal and annotation property keys, for example the code:

```
oi = find(id=tpl.any, label=icontains('OfIndividual'))
```

searches for all templates in **tpl** module with string 'OfIndividual' as part of a name, and assigns this set of URIs to **oi** variable.

For quick searches another Console command can be used to render templates in the active data source view. Executing **templates()** in the console, you will see group of all template definitions from the current data source (equivalent to ***All templates from data source*** menu command or Toolbar button). If used with a string argument – this command will show all templates with this substring as part of a name or of an URI (case-insensitive).

For example, use Console command:

```
templates('OfIndividual')
```

to see all templates in the active data source with string 'OfIndividual' as part of a name or URI.

Search by role restriction

Set of RDF triples describing role restriction for a particular role in a particular template is quite complex. Function:

<*module_name*>.tplrole(restriction, restricted_by_value_flag)

looks in **<*module_name*>** data source for templates with role (any role) restricted by **restriction** type or class, looking for restriction by value if **restricted_by_value_flag** is set to **True**.

For example, add ***IIP sandbox (templates)*** SPARQL endpoint through ***File*** menu, click project node in Project panel and assign to this endpoint **tpls** module name. Press ***All templates from data source*** button on the toolbar (SearchLan query for an endpoint is looking only for already downloaded data).

The command:

```
show(id=tpls.tplrole(part2.PossibleIndividual))
```

returns all templates where the role (some role) is restricted by PossibleIndividual type.

The command:

```
show(id=tpls.tplrole('http://rdl.rdlfacade.org/data#R247D172D2BE945A7814ADE2FF3A65A67', True))
```

returns the templates where the role (some role) is restricted by value with ACTUATOR DESCRIPTION class.

The command:

```
show(id=tpls.tplrole('http://www.w3.org/2001/XMLSchema#string'))
```

returns all templates where the literal role (some literal role) is restricted by XSD type *string*.

8.5. Search in verification and reasoning

While general-purpose OWL reasoning tools application to ISO 15926 data remains a problem, search language developed as part of .15926 Platform allows to build some special-purpose verification tools.

.15926 Editor built-in verification tests with rules encoded using SearchLan are described in **Volume 1. Getting Started**. These tests can be accessed from the Console using special value **wrong** documented above. Below you will find more tests implemented as queries or whole scripts which can automate some specific problem identification for PCA RDL. Even more such queries can be built from rules present in the standard itself, using mandatory requirements for data entities.

Other queries below are generated from template axioms. These are illustrating pattern recognition (already used in Editor's pattern search) usage for "template contraction" – reverse operation to template expansion, often necessary for pattern mapping process.

a. Quantified properties

The following set of queries closely follows structure of p7tpl:RealMagnitudeOfProperty template axiom. This is an attempt to verify important and voluminous part of RDL content required by the ISO 15926-2 approach to property modelling.

The first query shows all instances of Property:

```
show("all_prop", type=part2.Property)
```

The second query shows all properly quantified instances of Property:

```
show("quant_prop", type=part2.PropertyQuantification, hasInput=out,
hasResult=find(type=part2.RealNumber))
```

The third shows quantified instances of Property which have Scale properly defined:

```
show("scaled_prop", hasInput=out,  id=find(type=part2.Classification,
hasClassifier=find(type=part2.Scale),
hasClassified=out==check(type=part2.PropertyQuantification,
hasResult=find(type=part2.RealNumber))))
```

The query

```
show(id=all_prop - quant_prop)
```

returns all instances of Property which are not properly quantified by relating them to numbers.

The query

show(id=quant_prop - scaled_prop)

returns instances of Property which are quantified but Scale for the quantification is missing.

b. Object information models in RDL

Structure of this query closely follows p7tpl:IndirectPropertyScaleReal template axiom, but for classes on individuals, not for individuals. This query shows all classes of individuals which have in RDL a non-trivial object information model – non-empty set of indirect properties quantified by a single number (point value):

show(type=part2.ClassOfIndirectProperty, hasClassOfPossessor=groupby, hasPropertySpace=out==find(type=part2.Classification, hasClassifier=out, hasClassified=find(type=part2.PropertyQuantification, hasInput=out, hasResult=find(type=part2.RealNumber))))

c. Classification errors

This is the most complex test and the one which really uses Part 2 data model.

Part 2 contains many restrictions on class membership. Unfortunately they are mostly expressed in natural language and only part of them had found their way into the formal OWL model described on https://www.posccaesar.org/wiki/ISO15926inOWLPart2EntityMembership. Available formal restrictions are imported into the data model, augmented as described in **Volume 1**, and can be accessed in .15926 Editor environment, making it possible to automatically generate tests for correct class membership.

The following code cycles through all classifier classes in the RDL except instances of DocumentDefinition and RepresentationForm (for these two class membership restrictions are not obvious).

It takes the type of the classifier class and finds the set of all allowed types for its members, using internal representation of Part 2 membership restrictions encoded in the Editor. Then it takes the set of types of real members for classifier class analyzed and checks whether it is a subset of a set of allowed types. If not - there is a classification or typing mistake.

```
import iso15926.kb as kb
from graphlib import compact_uri, expand_uri, curi_head, curi_tail

def GetClassified(uri):
  result = set()
  curi = compact_uri(uri)
  name = curi_tail(curi)
  entry = kb.part2_itself["part2:" + name]
  classified = entry.get("classified")
  if classified:
    for v in classified:
      entry = kb.part2_itself[v]
      result.add(expand_uri(curi_head(curi)) + entry['name'])
  return result

def GetSubtypes(uriset):
  result = set()
  for uri in uriset:
    name = uri[max(uri.rfind('#'), uri.rfind('/'))+1:]
    if isinstance(getattr(part2.any, name).uri, set):
        subtypes = getattr(part2.any, name).uri-  set([getattr(part2, name).uri])
    result |= subtypes
  return result

all_classifiers=find(type=part2.Classification, hasClassifier=out)
docds= find(type=part2.Classification, hasClassifier=out==find(type=part2.DocumentDefinition))
rfrms= find(type=part2.Classification, hasClassifier=out==find(type=part2.RepresentationForm))

classifiers= all_classifiers-docds-rfrms
show(id=classifiers)

mist=set()
counter=0
for classifier in classifiers:
  counter+=1
  print(counter)
  classifier_type=find(id=classifier, type=out).pop()
  classified_type_must=GetClassified(classifier_type)
  classified_types_must= GetSubtypes(classified_type_must)
  classified_types_must  |= classified_type_must
  classified_types=find(id=find(type=part2.Classification, hasClassifier=classifier, hasClassified=out),
type=out)
  if classified_types&classified_types_must != classified_types:
    mist.add(classifier)
show(id=mist)
```

d. Specialization errors

This simpler query shows only classes of classes of individuals specialized from classes of relationships:

```
show(type=part2.Specialization,
hasSubclass=groupby==find(type=part2.any.ClassOfClassOfIndividual),
hasSuperclass=out==find(type=part2.any.ClassOfRelationship))
```

This query shows classes of relationships specialized from classes of classes of individuals:

```
show(type=part2.Specialization,
hasSubclass=groupby==find(type=part2.any.ClassOfRelationship),
hasSuperclass=out==find(type=part2.any.ClassOfClassOfIndividual))
```

8.6. Defining inheritance

Entities can inherit relationships and properties from their classifiers, superclasses and temporal wholes. Inheritance rules for ISO 15926 data model is rather complex and are not directly implemented in the .15926 Editor. However it is possible to write Python scripts specifying rules for the collection of inherited data and visualize the result.

For example, run the following code for PCA RDL file:

```
uri = find(label='PRESSURE MAINTAINING VALVE')
result = uri
while uri:
   uri = find(type=patterns.Specialization, superclass=out, subclass=uri)
   result |= uri
```

```
show('superclasses', id=result)
show('parts', type=patterns.Composition, whole=result, part=out)
show('properties', type=part2.ClassOfIndirectProperty, hasPropertySpace=out,
hasClassOfPossessor=result)
```

This script collects all superclasses for PRESSURE MAINTAINING VALVE and visualizes all superclasses (with PRESSURE MAINTAINING VALVE class itself), all classes of parts recorded for the class and its superclasses and all their properties. Such object information model for PRESSURE MAINTAINING VALVE is relatively rich: with inheritance from 13 superclasses it has 14 recorded relationships with possible parts (2 of them very generic), and 24 possible properties (only one of them generic).

8.7. Non ISO 15926 sources

With fully functional RDF viewer/editor at the core, .15926 Editor can parse and present any RDF compliant data set. Visualization of data, as well as type and role conventions, are specifically tailored for ISO 15926 data, but some analysis of other data sources is still possible. The Editor can easily handle large data sources which are difficult or impossible to explore in a majority of publicly available tools.

Scanner allows searches on non ISO 15926 data at least for labels and literal properties.

For example, download and unpack OpenCYC OWL knowledge base from http://www.opencyc.org/downloads. The file is more then 250 MByte (larger then current PCA RDL) and is almost impossible to work with on an average computer using Protégé (for example). Add it to .15926 Editor environment by dropping it on the Project panel

Search by label can be done from the search field at the top of a panel or from console, where logical and set conditions become available.

```
show(label=icontains("abnormal"))

show(label=icontains("abnormal")&icontains("cell"))
```

Search by universal key **literal** is also useful, allowing search for entities with particular substring in some literal property, without knowledge of property name or URI:

```
show(literal=icontains("medical"))
```

Logical condition for **literal property** key allows definition of restriction for some literal or annotation property of an entity, without knowledge of its name or URI. For example, query:

```
show(literal=icontains("medical")&icontains("private"))
```

returns entities which have some property with combination of substrings "medical" and "private" in its value.

Identify some annotation in the descriptions of found entities or directly on OpecCYC concept description pages. Study property semantics by opening its URI in browser, then register it in the property grid of a data source among other *Annotations*:

```
cycPrettyString http://sw.opencyc.org/concept/Mx4rwLSVCpwpEbGdrcN5Y29ycA
```

Now you can do the search by this property value:

show(cycPrettyString=icontains("biological"))

Register in *Roles owl:disjointWith* property which is still unknown to .15926 Editor:

owlDisjointWith http://www.w3.org/2002/07/owl#disjointWith

and execute the query:

show(label=icontains("cell"), owlDisjointWith=out)

All queries with **type**, **id**, **annotation** or **universal** keys are available for non ISO 15926 data source with some knowledge of its content. **Object property** and **pattern** queries can be used after some information on semantics of data model is gathered and used to customize project settings in the Editor.

8.8. SPARQL endpoint search

In .15926 Editor the main way to obtain data from SPARQL endpoint is search for substrings in entity labels or URIs performed via *search* box at the top of a data panel. Data entities found through such searches are bringing their relationships with them and by unfolding these entities volume of downloaded data can be increased. Downloaded data can be saved to local file by *Save as…* or *Save snapshot…* commands in *File* menu. Please note that unsaved downloaded data are deleted once you close current project or exit the Editor.

Even while downloaded data from endpoint are unsaved, you can use all power of SearchLan described above to query it. Of course you can never be sure that results of queries are representative or complete because search is restricted only to downloaded data.

Special functions to download data from SPARQL endpoints are included in Scanner API. This is just the beginning of SearchLan extension in this direction. To use most of these functions *Module name* has to be assigned to SPARQL data source in the project. For example, open PCA SPARQL endpoint and assign it a *Module name* **pca** in the project properties.

SPARQL query

Use SPARQL query forms **DESCRIBE** and **CONSTRUCT** to get an RDF graph from an endpoint. Only SPARQL queries which return RDF graph can be run from Console!

Returned RDF graph is added to the data already downloaded from an endpoint and visualised in data tree as a search result grouping node for further unfolding and exploration. Data becomes available for browsing and for further local analysis.

The simple syntax to obtain data from an endpoint and visualise it in an active panel is:

sparql('''*query text*''')

Please notice triple single quotation marks **'''** (as always in Python, triple double quotation marks **"""** can be used as well). According to Python syntax, preformatted really long strings containing several paragraphs can be put in triple quotation marks, allowing use of familiar-looking query forms.

For example, the following query for PCA endpoint data source shows all entities with "PUMP" substring in the designation:

```
sparql('''
PREFIX RDL: <http://posccaesar.org/rdl/>
PREFIX fn: <http://www.w3.org/2005/xpath-functions#>

describe ?class {
   ?class RDL:hasDesignation ?classDesignation .
   FILTER (fn:contains(?classDesignation, fn:upper-case("PUMP")))
 }
''')
```

Set of entities returned as subjects in RDF triples can be assigned to a Python variable using syntax similar to **show** command:

```
sparql('var', '''query text''')
```

To access one or multiple SPARQL endpoints in complex scripts or from extension, different syntax with module name should be used:

<*module_name*>.sparql_query('''query text''')

where <*module_name*> should be assigned to each processed endpoint. Use of this call is similar to the use of **find** command.

For example:

```
result = pca.sparql_query('''
PREFIX RDL: <http://posccaesar.org/rdl/>
PREFIX fn: <http://www.w3.org/2005/xpath-functions#>

describe ?class ?classDesignation  {
   ?class RDL:hasDesignation ?classDesignation .
   FILTER (fn:contains(?classDesignation, fn:upper-case("PUMP")))
 }
''')
show(id=result)
```

assigns the set of URIs of entities found to a variable named *result*.

Search for URI

To add data from endpoint for further local analysis without use of complex SPARQL queries, special function of SearchLan can be called:

<*module_name*>.find_uri(*value*))

where **value** is a single URI string or an URI set, specified either directly, or as a variable, or as a result of another query, or through a set operation on other sets. If you know some URI or URIs described in the endpoint, this function will download all their relations (triples where they are subject) and make them available for further search or save.

To visualize the results of the search you can use this function as a value for **id** key in SearchLan **show** function call. Try for PCA endpoint data source:

```
show(id=pca.find_uri("http://posccaesar.org/rdl/RDS1322684"))
```

This function can be used in Python scripts to look for data patterns starting from some known entity.

Search for RDS/WIP URIs

The next function works for PCA SPARQL endpoint only, as special RDF graph is kept at PCA endpoint containing references to the legacy RDS/WIP URIs (Rnnnnnn numbers). These references are linked through an RDF predicate *http://posccaesar.org/rdl/rdsWipEquivalent.*

You can make function call:

```
pca.find_uri_wip_eq(value)
```

where **value** is a single RDS/WIP URI or an URI set, specified either directly, or as a variable, or as a result of another query, or through a set operation on other sets.

The function returns a Python dictionary containing all found URI pairs in the format:

'http://rdl.rdlfacade.org/data#*Rnnnnn*' : 'http://posccaesar.org/rdl/*RDSnnnnn*'

For example, the call:

```
eq = pca.find_uri_wip_eq('http://rdl.rdlfacade.org/data#R52054275374')
```

returns:

```
{'http://rdl.rdlfacade.org/data#R52054275374': 'http://posccaesar.org/rdl/RDS1357739'}
```

The call:

```
eq = pca.find_uri_wip_eq(set(['http://rdl.rdlfacade.org/data#R52054275374',
'http://rdl.rdlfacade.org/data#R45527561750']))
```

returns:

```
{'http://rdl.rdlfacade.org/data#R45527561750': 'http://posccaesar.org/rdl/RDS357344',
 'http://rdl.rdlfacade.org/data#R52054275374': 'http://posccaesar.org/rdl/RDS1357739'}
```

You can combine this function with a previous one. The script:

```
eq = pca.find_uri_wip_eq('http://rdl.rdlfacade.org/data#R52054275374')
vl = eq['http://rdl.rdlfacade.org/data#R52054275374']
show(id=pca.find_uri(vl))
```

adds to the PCA endpoint data panel a node representing the class with legacy RDS/WIP identifier http://rdl.rdlfacade.org/data#R52054275374.

9. Builder.15926

9.1. Builder function calls

API functions of various .15926 Platform components are available in the Python console of the .15926 Editor and can be used in the Editor's extensions. For general information about Python console usage please refer to **Volume 1. Getting Started**, script examples can be found there also. Extensions of the Editor are described in **Volume 3. Extensions**. Use of scripting in mapping is also described in mapping specification for TabLan.15926 extension available with this distribution.

API of Builder.15926 component is described in this section of the documentation.

Builder API is available for local data sources only. Builder API functions could be called in a number of formats:

- Universal call:
  builder(type=<*module_name*>.TypeName, id=URI, edit= True/False, key1=value1, ..., keyN=valueN, delete= True/False)
- Create call:
  <*module_name*>.TypeName(key1=value1, ..., keyN=valueN)
- Change call:
  builder.role(id=URI, key1=value1, ..., keyN=valueN)
  builder.annotate(id=URI, key1=value1, ..., keyN=valueN)
- Edit call:
  builder.edit(type=<*module_name*>.TypeName, id=URI, key1=value1, ..., keyN=valueN)
- Delete call:
  builder.delete(value)
- Template call:
  builder.template(id=URI1, name=value, comment=string, super=URI2, delete= True/False)
- Template role call:
  builder.role(id=URI1, roleid=URI2, name=string, type=URI3, comment=string, value=URI4, delete=True/False)

Data created in the data source are the RDF/OWL serialization of ISO 15926 data compliant to Part 8. Builder API is not available for endpoints yet.

Please remember that changes to the data source through Builder function calls could not be undone!

If ***"Error: name '...' is not defined"*** error message is displayed in the console history panel – check whether indeed appropriate data source from a local file is open in your active panel or in a module you are addressing in your code statement.

Double vertical quotation marks **"** are used as string delimiters in examples below. According to Python language rules it is possible to use double or single quotation marks.

9.2. Universal call

Universal call for a **builder** has the following format:

> **builder(type=<*module_name*>.TypeName, id=URI, key1=value1, ..., keyN=valueN, edit=True/False, delete= True/False)**
>
> Where **key** is one of the following keys (key kinds):
>
> – **type** key;
> – **id** key;
> – ***role property*** keys;
> – ***annotation property*** keys;
> – ***literal property*** keys;
> – ***object property*** *keys;*
> – **edit** key;
> – **delete** key.

Builder call returns an URI of an instance created which can be assigned to a variable for further use.

9.2.1. Key **type**

Key **type** is used to specify the type for created entity. A value for **type** can be specified directly as a single URI string, as a variable whose value is an URI string, or in a format **<*module_name*>.TypeName**, where:

> **module_name** is a fixed module name **part2** or a module name for a template definition data source registered in the project;
>
> **TypeName** is an ISO 15926-2 entity type identified by CamelCase ID or a template name from the corresponding template definition data source;

This function creates an instance of a Part 2 entity type or an instance of a template specified in a template definition module. An instance is created in the data source opened in the active panel or in the data source identified via ***with context(…):*** statement.

For example:

```
uri1 = builder(type="http://rds.posccaesar.org/2008/02/OWL/ISO-15926-
2_2003#ArrangedIndividual", label="ind-13456")

uri2 = builder(type=part2.ArrangedIndividual, label="ind-63636")
```

The query:

```
show(id = set([uri1, uri2]))
```

checks that calls were executed successfully.

If ***p7tpl*** template definition module is present in the Project panel, the call:

```
uri3 = builder(type= p7tpl.ClassOfDefinition)
```

creates an unnamed instance of ClassOfDefinition template without any roles and assigns its URI to **uri3** variable.

The **builder** call will return an error if TypeName is not present in referred module.

If value for a **type** key is set to some arbitrary URI, an entity will be created with object property **rdf:type** with the specified value and without valid Part 2 type, which will not be compliant with ISO 15926 requirements. If **type** key value is set to non-well-formed URI, the Editor will try to interpret it as a fragment identifier for an URI in a default namespace, which may result in a corrupted data source.

9.2.2. Key **id**

Key **id** accepts any string value. If **builder** is called with **id** key the new entity gets the specified string value as its URI. No check is performed whether URI string is unique and well-formed or not. Non-well-formed URI may be interpreted by the Editor as a fragment identifier for an URI in a default namespace, leading to further mistakes. Data entity with URI which is not unique or well-formed can make data source RDF corrupted and uninterpretable!

If **id** key is not specified URI will be formed in the namespace defined as a *Namespace for new entities* for the edited data source according to the rules described in Volume 1. Getting Started: with fragment identifier made by concatenating prefix string (with default value "id") with the UUID compliant to RFC 4122 / ITU-T X.667 / ISO/IEC 9834-8.

To change default value of a prefix string for the current data source use **builder.set_uuid_prefix('*new prefix*')** command. For example, to make Editor generate RDS-UUIDs in the PCA RDL style use:

```
builder.set_uuid_prefix('RDS')
```

The prefix will be stored in project description file as data source property.

9.2.3. ***Role property*** keys

Role property keys are used in **builder** function calls to specify object roles of relational type instances – relationships, classes of relationships or template instances.

Role property keys include all roles of respective relational Part 2 type instances (for example, *hasClassifier*, *hasApproved*, *hasWhole* or *hasEnd1Cardinality*). Template role names are defined in template definition data source.

The value for a ***role property*** key can be specified directly as an URI string or as a variable whose value is an URI string. Remember that all SearchLan queries return URI sets and even one element set can not be accepted as ***role property*** key value.

For example, the following script creates two instances of PossibleIndividual and creates part-whole relationship between them:

```
uri4 = builder(type=part2.PossibleIndividual, label='A')
uri5 = builder(type=part2.PossibleIndividual, label='B')
uri6 = builder(type=part2.TemporalWholePart, hasPart=uri4, hasWhole=uri5)
```

If the last function call is changed to:

uri7 = builder(type=p7tpl.TemporalWholePart, hasPart=uri4, hasWhole=uri5)

template instance is created instead of Part 2 type instance.

If ***role property*** key for particular TypeName is misspelled or otherwise mangled, the key will be ignored. For template instance any role will be ignored if it isn't defined for this template. However if for some Part 2 relational type the key identifies some existing role which just doesn't belong to the type – an inappropriate role will be created. Please take care.

Instances of specialized templates with roles restricted by value are created with such roles already filled with appropriate value. An attempt to assign another value to such role at instance creation will be ignored.

9.2.4. ***Annotation property*** keys

Annotation property keys are used in **builder** function calls to specify annotation property values for entities created. Both instances of Part 2 types and template instances can possess annotation properties.

For each reference and project data source all annotations available to Builder are listed in the project property grid and in the data source property grid as *Annotations.* Each annotation property is registered there with a short name which is used as an ***annotation property*** key.

The value for an ***annotation property*** key can be specified directly as a string or as a variable whose value is a string. For example, function call:

**builder(type=part2.ClassOfIndividual, label="CONTAINMENT",
annCreationDate="2013-02-20")**

creates a new instance of ClassOfIndividual named CONTAINMENT and specified creation date.

If ***annotation property*** key is misspelled or otherwise mangled, the key will be ignored.

Sometimes generation of UUID compliant with RFC 4122 / ITU-T X.667 / ISO/IEC 9834-8 is required separately from URI generation (for example, to implement JORD ID Specification). Such UUID can be obtained through function call **builder.UUID('*prefix*')** .

For example, to assign a JORD ID Specification compliant R-UUID to a new entity (as a value of defaultRdsId property) add it to *Annotations* of a project or of data source as **key URI** pair (*defaultRdsId http://posccaesar.org/rdl/defaultRdsId*). The same R-UUID has to be a fragment identifier in the URI, which can be done with the following script:

def_id = builder.UUID('R-')

**builder(id='http://example.org/rdl/'+def_id, type=part2.ClassOfIndividual,
label="CONTAINMENT", annCreationDate="2013-02-20", defaultRdsId=def_id)**

9.2.5. ***Literal property*** keys

Literal property keys are used in **builder** function calls to specify literal property values for entities created. Some Part 2 type instances may possess literal properties (for example, Cardinality instances may have *hasMaximumCardinality* and *hasMinimumCardinality*, instances of six subtypes of ClassOfExpressInformationRepresentation may have *hasContent*). Template instances can also possess roles with literal values, to be filled with numbers, strings, dates and times, etc. (for example, template BeginningOfTemporalPart has role *valStartTime* restricted by type xsd:dateTime, template CardinalityEnd1MinMax has roles *valMaximumCardinality* and *valMinimumCardinality* restricted by type xsd:double).

Literal properties available to Builder are defined by Part 2 type system (available for review through *Data types* menu) or by template definitions.

The value for a ***literal property*** key can be specified directly as a string or as a variable whose value is a string. For example, function call:

```
builder(type=p7tpl.BeginningOfTemporalPart, hasPart=uri5, hasWhole=uri6,
valStartTime="2013-02-20 01:23:00.000000")
```

creates instance of BeginningOfTemporalPart template with two object roles and one literal role.

If ***literal property*** key is misspelled or otherwise mangled, the key will be ignored.

9.2.6. ***Object property*** keys

Object property keys are used in **builder** function calls to specify custom object roles used to record non-reified relationships in way not standard for reified ISO 15926 RDF representation.

Unlike ***role property*** keys, custom ***object property*** keys should be registered as *Roles* in the property grid of a project or of a data source before further use. Short names registered in the *Roles* are used as ***object property*** keys.

The value for an ***object property*** key can be specified directly as an URI string or as a variable whose value is an URI string. Remember that all SearchLan queries return URI sets and even one element set can not be accepted as ***object property*** key value.

To execute an example below, find custom object property **subClassOf** property in properties panel for the project or for the data source. If nor found register it yourself –find *Roles* row in project properties, double click it and add a string:

```
subClassOf http://www.w3.org/2000/01/rdf-schema#subClassOf
```

The following script creates two instances of PossibleIndividual and creates **subClassOf** property to record relationship between them:

```
uri6 = builder(type=part2.Class, label='classA')
uri7 = builder(type=part2.Class, label='classB', subClassOf= uri6)
```

9.2.7. Key **edit**

If **builder** is called with **id** key value of an existing URI (specified directly or by a variable) – the call will edit the entity with specified URI. The nature of changes depends on the value of **edit** key. All keys (t**ype**, **role**, ***literal***, ***annotation*** and ***object property*** keys), their values and usage in calls with **edit** key are subject to all the rules described above.

If particular **role** or ***literal property*** key is not allowed by the type of an edited entity, it will be ignored, except in the situation when type is changed simultaneously (see below)

If **id** has existing URI as value and **edit** key is set to **False** or omitted, new roles, literal, annotation or object properties will be added as specified by keys present, even if such roles or properties already exist. Such call of **builder** literally **adds** new roles and properties to an existing entity, it doesn't check whether such roles or properties already exist and doesn't change values of existing roles or properties, but creates them again with new values.

Additional value of ***rdf:type*** can be assigned by using **type** key, for example. Multiple annotations of the same type can be created as well.

If **id** has existing URI as value and **edit** key is set to **True**, new values for existing roles, literal, annotation or object properties will be added, replacing old values of roles and properties and deleting multiple values of the same role or property, if any. If there were no such roles or properties – they will be just added, of course.

If **edit** key is set to **True** and key **type** is set to a new value, all roles specific to old type (as defined by Part 2 data model or template definition) will be deleted. If the same function call contains role keys specific to a new type – they will be added as specified.

If **edit** key is set to **True** and key **type** is set to the special value ***None***, all ***rdf:type*** properties and all roles specific to these types are deleted.

If **edit** key is set to **True** and any other key (except **type**) is set to the special value ***None***, all values of the corresponding role or property will be deleted.

9.2.8. **Delete** key

If **delete** key is set to **True** and **id** key value is an URI string, a variable whose value is an URI string, or an URI set – all entities with these URIs are deleted. Effect of a ***delete*** function can not be undone!

9.3. ***Create*** call

The following simplified call for **builder** function is available for creation of new entities:

<*module_name*>.TypeName(key1=value1, ..., keyN=valueN)

or

<*module_name*>.TemplateName(role1, … roleM, key1=value1, ..., keyN=valueN)

This call is equivalent to the **builder** call with **type=<*module_name*>.TypeName.**

Role keys, ***literal*****,** ***annotation*** and ***object property*** keys, their values and usage are subject to all the rules described above. Key list may be empty, but do not forget to write the parentheses **()**!

The function call:

uri8 = part2.ClassOfRelationship()

creates unnamed instance of the ClassOfRelationship.

The query:

show(id = uri8)

checks that this call was executed successfully.

If p7tpl template definition module is present in the Project panel, the call:

```
uri9 = p7tpl.ClassOfDefinition()
```

creates an instance of ClassOfDefinition template without roles.

If you are an experienced Python programmer, you can use built-in Python function ***getattr()*** in your scripts – it allows to call Builder functions if types you need are available as strings or string variables. For example, the following script is equivalent to the previous example:

```
type_str = "ClassOfDefinition"
uri11 = getattr(p7tpl, type_str)()
```

In **builder** call of this kind **role property** keys may be omitted for template instances, as templates have a fixed role order. ***Role property*** keys should be omitted for all template roles simultaneously. Function call should start with a list of role URIs, and all other keys may be placed after this list.

For example, one of example function calls above can be executed as:

```
uri10 = p7tpl.TemporalWholePart(uri4, uri5)
```

9.4. Change call

The following ways to call **builder** are preserved to keep compatibility with earlier versions of Builder API.

9.4.1. ***Add role*** call

Add role call is available as:

```
builder.role(id=URI, key1=value1, ..., keyN=valueN)
```

This call is equivalent to the **builder** call with **edit=False,** and **key** is one of the ***role property*** keys, ***literal property*** keys or ***object property*** keys.

Add role call adds object or literal property to an already existing instance of Part 2 type or to a template instance. An edited entity is identified by **id** key value - an URI string or a variable whose value is an URI string.

Role property keys, ***literal property*** keys and ***object property*** keys, their values and usage in an ***add role*** function call are subject to all the rules described above.

Please remember that ***add role*** function literally **adds** new properties to an existing entity, it doesn't check whether such properties already exist and doesn't change values of existing properties, but duplicates them with new values.

9.4.2. ***Add annotation*** call

Add annotation call has the following format:

builder.annotate(id=URI, key1=value1, ..., keyN=valueN)

This call is equivalent to the **builder** call with **edit=False,** and **key** is one of the ***annotation property*** keys.

Add annotation call adds annotation property to an already existing instance of Part 2 type or to a template instance. An edited entity is identified by **id** key value - an URI string or a variable whose value is an URI string.

Annotation property keys, their values and usage in an ***add annotation*** function call are subject to all the rules described above.

Please remember that ***add annotation*** function literally **adds** new properties to an existing entity, it doesn't check whether such properties already exist and doesn't change values of existing properties, but duplicates them with new (or old) values.

9.5. Edit call

Edit call is available as:

builder.edit(type=<*module_name*>.TypeName, id=URI, key1=value1, ..., keyN=valueN)

This call is equivalent to the **builder** call with **edit=True.** ***Type*** key, ***literal*****,** ***annotation*** and ***object property*** keys, their values and usage are subject to all the rules described above!

9.6. Delete call

Delete call has the following format:

builder.delete(value)

where **value** is an URI string, variable whose value is an URI string, or an URI set.

Delete call deletes all entities whose URI's are passed to it. Effect of a ***delete*** call can not be undone!

9.7. Template definition builder

Special functions are defined in Builder API for creation and editing of template definitions in template definition data sources. Template definition creation functions are useful in conversion of template definitions from other formats. For example, these functions are used in open-source built-in extension for import of template definitions from spreadsheet used to populate iRINGTools software (see **Volume 3. Extensions** for details).

If URI of existing template or role is referenced in function calls described below, template or role with this URI will be changed as specified by the call.

If you are modifying templates and creating instances of templates in the same script (quite exotic situation), new or modified definitions become available for instance creation only after the function call ***<module_name>*.update_templates()** .

For example:

```
builder.template(name = 'NewTemplate')
mmttpl.update_templates()
inst = mmttpl.NewTemplate()
```

9.7.1. Template call

Builder API function call to create or edit template definition in template definition data source has the following format:

builder.template(id=URI1, name=value, comment=string, super=URI2, delete= True/False)

where keys are:

id – template URI, mandatory for deletion or editing, if missing – URI will be generated automatically according to data source settings as described above;
name – template name;
comment – comment field;
super – parent template URI for creation of specialized templates;
delete – if **True** deletes a template specified by **id** (Builder will not control deletion of parent templates for specialized templates in a data source).

This call returns URI of a template created.

9.7.1. Template role call

Builder API function call to create or edit template role in template definition has the following format:

builder.template_role(id=URI1, roleid=URI2, name=string, type=URI3, comment=string, value=URI4, delete=True/False)

where keys are:

id – template URI;
roleid – role URI, mandatory for deletion or editing, if missing – URI will be generated automatically as described above according to data source settings (in a data source all roles with the same name will receive the same URI, if URI is not directly set by this key);
name – role name, mandatory;
type – role type restriction as URI or in **part2.TypeName** format;
comment – comment field;
value – entity URI for role restriction by value in specialized templates (overrides **type** key value);
delete – if **True** deletes a role specified by **roleid.**

This call returns URI of a role created.

Builder remembers the URI of last template created, therefore **id** key can be omitted if template role calls are made immediately after template creation call (this feature doesn't work if calls are performed from Python console one by one!).

***