2021

The UCO/CASE Validation Toolkit

DR. STANLEY A BERNSTEEN

Table of Contents

The UCO/CASE Validation Toolkit

1 Introduction

The UCO/CASE Validation Toolkit provides the capability to validate JSON-LD data files against a turtle-file based ontology such as the Unified Cyber Ontology (***UCO***) and Cyber-Investigation Analysis Standard Expression (***CASE***) ontologies.

1.1 Ontologies

An ***ontology*** is representation of categories, properties, and relationships within a subject area. It can be expressed as a set of statements of the form "category X has a property called Y whose value is Z", or simply the triples (X, Y, Z).

There are several different syntaxes for representing ontologies. This toolkit applies to ontologies expressed in the Web Ontology Language (***OWL***) language, as described at https://www.w3.org/OWL, and represented in the file using the Terse Resource Description Framework Triple Language (***turtle***) syntax, as described at https://www.w3.org/TR/turtle.

1.2 Data

Data ***conforms*** to an ontology if it describes objects that are instances of categories of the ontology and is consistent with the relationships and constraints described in the ontology. A data object can be expressed as "item X is an instance of category Y and has properties P_i with values V_{ik}", where each value is either a pointer to another data object or a literal value.

There are several different syntaxes for representing data. This toolkit applies to data represented in JavaScript Object Notation for Linked Data (***JSON-LD***), as described at https://JSON-LD.org/spec/latest/JSON-LD.

1.3 Validation

The purpose of the UCO/CASE Validation Toolkit is to check whether a given set of data conforms to a given ontology, and to call out the places where it does not. It also identifies inconsistencies within the ontology itself.

Although this toolkit is designed to work for any ontology, it is particularly intended to work for UCO up to version 0.5, as described at https://github.com/ucoProject/uco, and the CASE ontology up to version 0.2, as described at https://github.com/casework/CASE.

2 Toolkit Requirements

The UCO/CASE Validation Toolkit shall validate that data conform with an ontology. It shall input an ontology expressed as turtle files, data formatted as JSON-LD files then output a report indicating inconsistencies within the ontology and places where the data does not conform with the ontology. The output messages shall include the location of the problem within the input file and a description of the problem.

Inconsistencies identified in the ontology shall include but not be limited to

- Contradictory constraints.
- Properties with multiple ranges.
- Properties with no ranges.
- Non-OWL classes.
- Use of unsupported features (see below).

Conformance issues identified in the data shall include but not be limited to

- Items with @type not in the ontology.
- Items with multiple @types.
- Items missing properties required by the ontology.
- Items with properties violating constraints specified in the ontology.
- Items containing invalid literals.
- Items linking to non-existent items.

The most-used features of the OWL language in ontologies shall be supported. Unsupported rarely used features shall include but not be limited to

- Lists explicitly using olo#slot and olo#item.
- Non-XSD #onDataRange constraints, such as pattern:PatternExpression and core:Confidence
  - Exception: vocabularies (linked lists of vocabulary terms) *shall* be supported.

Shape Constraint Language (***SHACL***) expressions (see below) shall not be supported.

The “best-practices” features of JSON-LD in data shall be supported. Unsupported JSON-LD constructs shall include but not be limited to

- Multiple @contexts. Each document should start with a single global @context section.
- Complex @contexts. Prefixes in the @context section should map to IRIs, not to structures.
- Multiple documents in one file. JSON-LD files should contain a single document.
- Links to items in other JSON-LD files.
- Poorly formatted files. Valid but “ugly” JSON-LD is not supported.
  - Exception: to accommodate legacy data, items with blank prefixes *shall* be supported, even though blank-prefixed data is invalid JSON-LD.

Starting from Version 0.6, the UCO ontology will be using SHACL constraints instead of OWL constraints. SHACL is an extension to the turtle file syntax that provides a clearer and better way to specify ontology constraints than OWL does. There are many additional tools that validate data for SHACL-based ontologies, such as Apache Jena, and rdflib/pySHACL. This Toolkit does not support SHACL.

3 User Document

3.1 Installation on *nix Systems

Assuming access to MITRE's gitlab, install the Tookit on a *nix box as follows.

- Create empty working directory.
  - `$ mkdir foo`
  - `$ cd foo`
- Copy source files into the working directory.
  - `$ git clone https://gitlab.mitre.org/sbernste/enhanced-case-toolkit.git`
  - If external to MITRE, a local copy of the source files will be required.
- In the tookit directory, create a python3 virtual environment.
  - `$ cd enhanced-case-toolkit`
  - `$ virtualenv --python=python3 venv`
- Activate the virtual environment.
  - `$ source venv/bin/activate`
- Install the source into the virtual environment.
  - `$ pip install --upgrade pip`
  - `$ pip install --editable src`
- Run the toolkit commands as described below.
- When done, deactivate the virtual environment.
  - `$ deactivate`

3.2 Installation on Windows Systems

Assuming access to MITRE's gitlab, install the Toolkit on a windows box using powershell as follows.

- Create empty working directory.
  - `> mkdir foo`
- Copy source files.
  - `> cd foo`
  - `foo> git clone https://gitlab.mitre.org/sbernste/enhanced-case-toolkit.git`
  - If external to MITRE, a local copy of the source files will be required.
- Create virtual environment.
  - `foo> cd enhanced_case_toolkit`
  - `foo\enhanced-case-toolkit> virtualenv --python=python3 venv`
- Activate the virtual environment.
  - `foo\enhanced-case-toolkit> venv\Scripts\activate`
- Install the source into the virtual environment.
  - `(venv)foo\enhanced-case-toolkit> pip install --editable src`
- Run the toolkit commands as described below.
- When done, deactivate the virtual environment.
  - `(venv) foo\enhanced-case-toolkit> deactivate`

3.3 Using the Validator

The validator script validates JSON-LD data against an ontology.

To validate one or more JSON-LD files against an ontology, enter this command:

```
validate ontology_path [jsonld_path...]
```

where:

`ontology_path` is the full path to the directory containing the ontology turtle files (*.ttl), or the path to a serialized ontology file (*.pkl). If a turtle file directory is specified, the validator also checks the ontology for internal consistency.

`jsonld_path` is the full path to zero or more JSON-LD data files (*.json) to validate against the specified ontology, or the path to serialized JSON-LD files (*.pkl). If no paths are specified, the validator still checks the ontology for internal consistency.

3.4 Using the Data Upgrader

The data elevator script converts data written for Version 0.4 of the UCO/CASE ontology to Version 0.5.

Data files written for UCO/CASE ontology version 0.4 are not compatible with version 0.5. The changes include:

- **Relocation of investigation classes**. The investigation classes moved from https://unifiedcyberontology.org/ontology/uco/investigation to https://caseontology.org/ontology/case/investigation.
- **Renaming of classes**. Many classes were renamed by appending "Facet" to their name. For example, the Account class was renamed AccountFacet.
- **Splitting of vocabulary classes**. Some of the vocabulary classes were removed from the https://unifiedcyberontology.org/ontology/uco/vocabulary namespace space and moved into the https://caseontology.org/ontology/case/vocabulary namespace. There are now two separate vocabulary spaces. Sample data now needs to distinguish which vocabulary belongs in which space when implementing their context.

The full set of changes are explicitly listed at the UCO website https://unifiedcyberontology.atlassian.net/wiki/spaces/OC/pages/719716353/UCO+Version+0.5.0+Release+Notes

To convert data from version 0.4 to 0.5, cd to the source directory and enter this command:

```
v4to5 [-o output_filepath] input_filepath
```

where:

`input_filepath` is the full path to the version 0.4 JSON-LD data file.

`output_filepath` is the full path to the converted version 0.5 JSON-LD data file. If not specified, the file is written to stdout.

3.5 Using the Serializer

The serialize script serializes ontologies and JSON-LD data files. Invoking the validator with serialized files saves the time it takes to load and process the file.

To serialize an ontology or a JSON-LD file, enter this command:

```
serialize [-c comment] [-o output_filepath] input_filepath
```

where:

`input_filepath` is the full path to the directory containing ontology turtle files, or the full path to a JSON-LD file.

`output_filepath` is full path to the output serialized file. If not specified, it is the same as the input filepath plus ".pkl".

`comment` is a description to be included in the serial file, enclosed in quotes. If not specified, the user will be prompted to enter the description. Since the description is read from stdin, it can be redirected from a file.

To read the comment in a serialized file, cd to the source directory and enter this command:

```
describe serialized_filepath
```

WARNING: *Serialized files contain a snapshot of its source data as it was at the time it was serialized*.

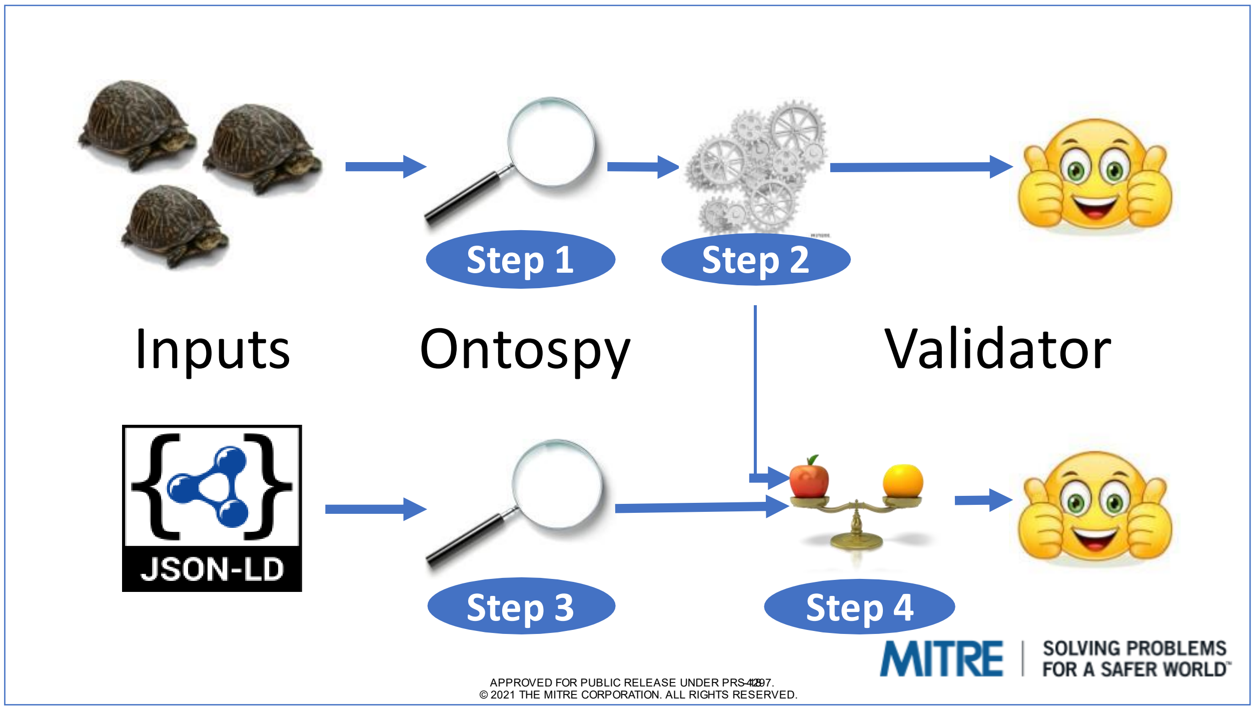
Serialized ontologies contain the ontology *and its error reports* as they were at serialization time. If the ontology changes, the changes are not reflected in the serialized file. If the toolkit software changes, the changes are not reflected in the error messages that were serialized with the file.

Similarly, serialized JSON-LD files contain the JSON-LD data as it was at serialization time. If the JSON-LD file changes, the changes are not reflected in the serialized file.

In both cases, if ontospy is upgraded, the serialized files will not reflect the upgrade.

4 Technical Overview

The UCO/CASE Validation Toolkit performs these four steps:


Step 1
Step 2
Inputs
Ontospy
Validator
JSON-LD
Step 3
Step 4
MITRE | SOLVING PROBLEMS FOR A SAFER WORLD
APPROVED FOR PUBLIC RELEASE UNDER PRS-4297.
© 2021 THE MITRE CORPORATION. ALL RIGHTS RESERVED.

Step 1. Ingest ontology. Use ***ontospy*** to ingest the ontology turtle files and decompose them into their component parts.

Step 2. Identify property constraints. Process the OWL-based property constraints in the ontology.

Step 3. Ingest JSON-LD data. Use ***ontospy*** to ingest the JSON-LD data file and decompose it into its component parts.

Step 4. Validate the data. Compare the JSON-LD data with the ontology and report constraint violations.

4.1 Ingesting the Ontology

Ontospy is an open-source python application for working with vocabularies encoded in the Resource Description Framework (***RDF***) family of languages. It recognizes data expressed as turtle files or as JSON-LD and understands OWL and SHACL expressions.

The validator uses ontospy to read an ontology expressed as turtle files. Ontospy decomposes the turtle statements into RDF triples, the (X, Y, Z) described in the introduction. Ontospy also cross-correlates the classes and properties and computes class hierarchy, property domains and ranges, and other derived information in convenient structures. These are listed in Appendix A.

Unfortunately, ontospy does not compute the class constraints in the turtle files; it does not process OWL-based class constraints. The validator needs these constraints to validate data, so it computes them in Step 2.

4.2 Identifying Property Constraints

OWL is awkward about expressing property constraints; it resorts to introducing auxiliary classes with the desired property constraints and declaring the constrained class as a subclass of the auxiliary classes. Ontospy does not process OWL-based property constraints, but it does reduce them to RDF triples. The validator derives the property constraints for each class from the RDF triples and provides the information in a convenient data structure.

Next, the validator invokes inheritance. It uses the class hierarchy to pass property constraints from parent to child class, so that each class's set of constraints includes all inherited constraints. The validator reports conflicting property constraints found during the process, and checks for untyped properties, malformed hierarchies, and other ontology specifications that are syntactically correct but semantically inconsistent.

It takes about two minutes to ingest the UCO/CASE ontology, identify the property constraints, and do the consistency checks. This delay can get annoying during the development cycle, so the UCO/CASE Validation Toolkit provides the capability to serialize the ingested ontology along with its property constraints and consistency reports. Ingesting the serialized file only takes about a second and is therefore more convenient to work with. The serialization module is described later in this document.

4.3 Ingesting JSON-LD Data

The validator uses ontospy to read data from JSON-LD data files. Ontospy decomposes the JSON-LD statements into RDF triples suitable for comparison with the ontology specifications.

Turtle file syntax allows Uniform Resource Identifiers (***URI***s) to contain empty prefixes, but JSON-LD syntax does not. Many software packages, including ***Ontospy***, do not accept JSON-LD files containing empty-prefixed URIs. If the validator encounters a JSON-LD file containing empty-prefixed URIs, it scans the file and replaces the blank prefixes with a prefix that is not used anywhere else in the file. The preprocessor's empty-prefix scanner is not very sophisticated; it relies on the JSON-LD file being formatted according to "best practices": one item per line, etc.

One of the validator's requirements is that errors in the data shall be identified by its location in the file. When ontospy ingests the JSON-LD file, it loses all the original formatting information. To preserve this information, the validator implements a preprocessor that reads the JSON-LD file and textually embeds the line number of each @type statement into the type. Since every data item in the file should have a @type, this effectively maps every item to its line number. After ontospy reads the JSON-LD file with the modified @types, the validator postprocesses ***ontospy's*** RDF triples by extracting the line numbers from the @types and reverting them to their original value, remembering which @type statement had which line number.

There is an ambiguity when the object of a predicate in the JSON-LD file contains string that is really a URI but there are no constraints for the predicate. When this happens, ontospy cannot tell whether the string is a literal or an URI that is supposed to link back to another object in the file. In this case, ontospy always assumes that it is a literal string. The validator's postprocessor detects these misidentified strings by searching the RDF triples for literals that "look like" a URI with a prefix that is defined in the @context and replacing those literals with URI objects.

4.4 Validating the Data.

The validator validates the data by comparing its RDF triples from Step 3 with the ontology description and constraints from Steps 1 and 2. If it finds any triple to violate the constraints, the validator outputs a message describing the violation.

Section 5.1 describes the data validation algorithm in detail. In summary, data validation proceeds in four steps. First, sort the RDF triples from ontospy (subject, predicate, object) by subject, where each subject corresponds to one of the data items in the JSON-LD document. Then, identify the ontology class that this data item belongs to (defined in one of the triples) and its properties and values (defined in the rest of the triples). Next, compare each property and value with the property constraints compiled for the data item's class. Finally, make sure literal values are internally consistent, e.g., values that say they are integers are really integers.

5 Implementation Details

This section describes the modules comprising the Validator and the algorithms they implement.

5.1 Validation

Module **validator.py** implements the data validation algorithm.

To validate JSON-LD case data against an ontology, the first step is to use Ontospy to decompose the JSON-LD data into a set of RDF triples, (***subject***, ***predicate***, ***object***).

Imagine putting this list of triples into an Excel spreadsheet and sorting them with a primary sort on subject and a secondary sort on predicate. We can express this sorted list of triples as a structure that relates each subject to its predicates, and each of the subject's predicates to its objects. We call this the subject-predicate-object dictionary, or the ***spo dictionary***. So **spo[subject]** is a list of the subject's predicates and their objects, and **spo[subject][predicate]** is a list of the objects for one of the subject's predicates. To validate the sample data against an ontology, we validate the properties of each subject in the ***spo dictionary*** against the ontology.

Each subject ***S*** is an instance of some class ***C*** in the ontology. The ***spo dictionary*** declares this class as the **RDF.type** predicate; so ***C*** is **spo[*S*][RDF.type]**.

We can now check that the rest of the data for ***S*** conforms to the constraints of class ***C*** in the ontology. The values ***spo[S][P]*** (except for ***P*** = ***RDF.type***) describe values of property ***P*** of subject ***S***. The ontology may specify the allowable number of values of each property ***P*** and/or the value's required type. It is easy to validate that ***spo[S][P]*** has a valid number of values; validating value type is a bit trickier.

Values come in two varieties: they are either pointers to other data items (subjects in the ***spo dictionary***) or instances of **Literal**. If a value is a pointer to subject ***S'***, its type is **spo[*S'*][RDF.type]**. If the value is an instance of **Literal**, its type is explicitly specified inside the **Literal** object. Once we have determined the value's type, it is easy to validate that the ontology allows this type of value for this property.

The final step is to validate the instances of **Literal**. Each **Literal** object encapsulates its type and its value as a string, and the string needs to be consistent with the type. **Literal** types come in two varieties: ***simple types*** and ***datatypes***. A ***simple type***, such as string or integer; we use python's built-in XSD validator to validate simple types. A ***datatype*** is a subset of a simple type, such as integers between 0 and 100, dates in the year 2020, or strings belonging to a vocabulary. The toolkit contains a validation function for each supported ***datatype***.

5.2 Preprocessing/Postprocessing

Module **precondition.py** implements the preprocessing and postprocessing described in section 4.3.

The JSON-LD data preprocessor function is the most fragile part of the validation procedure. It implements simple pattern matching and substitution algorithms that assume the JSON-LD file is laid out in a reasonable way. Although it works on our sample input files, it is easy to defeat the preconditioner with syntactically correct but ugly input data. The preprocessor assumes:

- @type statements define a single type (not a list) on single line with nothing else in the line
- The JSON-LD file contains only one @context section, and it is global to the file.
- If the @context section defines an empty context, it appears on its own line and consists of a string that starts with *exactly* this: `"":<s>"<h>://`, where **`<s>`** is zero or more spaces, **`<h>`** is either http, https or file, and there are *exactly* three double-quotes as shown.
- Any subject or object that has an empty prefix looks *exactly* like this: `":<x>"`, where **`<x>`** is at least one letter, number, hyphen or underscore, and there are *exactly* two double-quotes as shown.

First, the preprocessor must remove empty prefixes to turn the file into a valid JSON-LD file. It looks for an empty prefix declaration in the file's @context section. If it finds one, it searches the file for all possible three-character prefixes (sequences of three characters followed by a colon), chooses a three-character prefix not in that set, and affixes the chosen prefix to all the empty-prefixed subjects and objects in the file.

Next, the preprocessor looks for @type statements in the JSON-LD file and appends **`_LINE_`*****`n`*** to each type, where ***`n`*** is the line number (any number of digits) of the @type statement. (It would have made more sense to tag the @id statements instead, but that breaks ***ontospy***.)

When validator presents the preprocessed file to ***ontospy***, ***ontospy*** decomposes it into RDF triples. Those triples are passed to the postprocessor.

The postprocessor operates directly on the RDF triples. First it extracts the line number ***`n`*** from the subject and object graph nodes appended with the **`LINE_`*****`n`*** suffix, and removes the suffix, building a mapping from the RDF triple node to its line number for later reference.

Next, the postprocessor needs to address an ambiguity inherent in JSON-LD. The ambiguity occurs when the object of a predicate contains string that is really an IRI, but there are no constraints for the predicate. It is impossible to tell whether it is a simple literal or an IRI that is supposed to link back to another object in the file, and ***ontospy*** assumes it is a literal. The postprocessor searches the RDF triples for literals that "look like" an IRI, using a prefix that is actually defined in the context, and replaces those Literals with URI objects. False identification of URIs is possible but unlikely.

5.3 Working with Ontologies

Module **ontology.py** implements the extraction and collection of OWL-based property constraints and other information from ***ontospy*** after ***ontospy*** has read an ontology and defines the **Ontology** object to contain this information.

When ingesting an ontology from turtle files, the first step is to allow ***ontospy*** to read the turtle files. The second step is to derive the class and datatype constraints from the OWL-based class constraints. The third step is to implement class inheritance, adding to each class's constraints the constraints of its parent classes. The final step is to collect the property ranges from ***ontospy*** and check for inconsistencies between the property ranges declared as part of OWL class constraints and those declared elsewhere in the turtle files.

Note that the validator parses only OWL-based constraints, not SHACL constraints. This validator does not handle SHACL.

5.3.1 OWL Class Constraint Parser

Each class in the ontology is either an **OWL.Class** or an **RDFS.Datatype**. The purpose of the OWL-constraint parser is to associate a **ClassConstraints** or **DatatypeConstraints** object with each class in the ontology.

Ontospy provides the RDF triples for each class in the ontology. The RDF triples define the constraint conditions for each ontology class, and the sum of the triples for each class defines a **ClassConstraints** or a **DatatypeConstraints** object.

For each **OWL.Class** class, the ontology module calls a function in the **class_constraints** module that uses the RDF triples to build a **ClassConstraints** object. Each **ClassConstraints** object contains a set of **PropertyConstraints** objects, each of which describes the cardinality and/or range constraints for a single property. See section 5.7 for more details.

For each **RDFS.Datatype** class, the ontology module uses the RDF triples to build a **DatatypeConstraints** object, which describes the ***datatype*** class. A ***datatype*** class defines a simple object, such as an integer or a string, or a subset of a simple object, such as integers between 1 and 100, or strings from a vocabulary list; the **DatatypeConstraints** object contains a function called **validate()** that validates strings against its data type. Although several kinds of datatype are allowed, the validator currently supports only one type: the ***vocabulary***, which defines a list of valid strings. See section 5.6 for more details.

5.3.2 Inheritance

The property constraints inheritance algorithm applies only to **ClassConstraints**. It is based on the OWL-constraints mantra: **if a parent class and a child class each define constraints for a property, the child's constraints must be the same as or a subset of the parent's.** Based on this mantra, the inheritance algorithm implements these rules:

- If the child class has no constraint, it inherits its parent's constraints.
- If the child's constraint is the same as or a subset of the parent's, there is no change.
- If the child's constraint is a proper superset of the parent's, there is still no change, but the validator reports an Ontology Error.

Specifically, for property constraints, inheritance of cardinality and property range constraints is defined in the following tables:

Minimum Cardinality Inheritance

Child Min card	Parent Min card	Condition	Updated child Min card	Change	Error	Comment
None	None		None	No	No	No change
None	Y		Y	YES	No	Child inherits parent's constraint
X	None		X	No	No	Child keeps its constraint
X	Y	X > Y	X	No	No	Child keeps its subset constraint
X	Y	X = Y	X	No	No	Constraints agree
X	Y	X < Y	X	No	ERROR	Child's constraint not a subset

Maximum Cardinality Inheritance

Child Max card	Parent Max card	Condition	Updated child Max card	Change	Error	Comment
None	None		None	No	No	No change
None	Y		Y	YES	No	Child inherits parent's constraint
X	None		X	No	No	Child keeps its constraint
X	Y	X < Y	X	No	No	Child keeps its subset constraint
X	Y	X = Y	X	No	No	Constraints agree
X	Y	X > Y	X	No	ERROR	Child's constraint is not a subset

Since the minimum and maximum cardinality inheritance rules are independent, it is possible that min_cardinality ≥ max_cardinality after both are inherited. The algorithm checks for this condition.

Property Range Inheritance

Child range	Parent range	Condition	Updated child range	Change	Error	Comment
None	None		None	No	No	No change
None	Y		Y	YES	No	Child inherits parent's constraint
X	None		X	No	No	Child keeps its tighter constraint
X	Y	X is subclass of Y	X	No	No	Child keeps its subset constraint
X	Y	X = Y	X	No	No	Constraints agree
X	Y	Y is subclass of X	X	No	ERROR	Child's constraint is not a subset
X	Y	X unrelated to Y	X	No	ERROR	Child's constraint is not a subset

5.3.3 Property Range Consistency
The final step is to check the consistency of property ranges in the ontology. This step is necessary because property ranges for the same property can be defined in multiple places in the ontology. The function **check_range_consistency()** in **ontology.py** implements this consistency check.

5.4 Working with JSON-LD Data
Module **casedata.py** extracts the information JSON-LD data files required to validate it and defines the **CaseData** object to contain this information.

First, it uses the **precondition()** function described in section 5.2 to fix syntax issues in the JSON-LD file and to embed line numbers in the objects defined in the file. Then is uses ***ontospy*** to reduce the JSON-LD file to its component RDF triples. Finally, it uses the **postcondition()** function to extract the embedded line numbers from the RDF triples and associate the objects defined in the JSON-LD file with their line number, and stores this information in the CaseData object.

5.5 Serialization
The **serializer.py** implements the toolkit serialization methods for serializing ontologies and JSON-LD data files. The ***serialize()*** function in the **Ontology** and **CaseData** classes each call the serializer.

Each serialized file is a binary file that contains three binary components:

- The 8-byte identifier ("magic number"): "**ontology**" or "**casedata**". This allows the software to quickly determine whether a file has been serialized by this toolkit.
- The python-pickled metadata dictionary. This contains a user-supplied description of the original file, and the date and time it was serialized. It also contains the source filepath and its md5 hash, allowing the software to determine whether the source has changed since it was serialized. For ontologies, it also contains the md5 of each of the source turtle files, allowing software to determine which individual turtle files have changed.
- The python-pickled data (the **__dict__** component) of an Ontology or CaseData object.

This file construction allows software to examine a serialized file's metadata without having to read the entire file. The serializer module contains functions to get the individual components cited above.

WARNING: *Serialized files contain a snapshot of its source data as it was at the time it was serialized*.

Serialized ontologies contain the ontology *and its error reports* as they were at serialization time. If the ontology changes, the changes are not reflected in the serialized file. If the toolkit software changes, the changes are not reflected in the error reports in the serialized file.

Similarly, serialized JSON-LD files contain the JSON-LD data as it was at serialization time. If the JSON-LD file changes, the changes are not reflected in the serialized file.

In both cases, if ontospy is upgraded, the serialized files will not reflect the upgrade.

5.6 Datatype_Constraints
A ***datatype*** is a simple object, such as an integer or a string, or a subset of a simple object, such as integers between 1 and 100, or strings from a vocabulary list. The XSD Validator module handles validation of simple datatype; the **Datatype_Constraints** module supports the subset objects. Module

datatype_constraints.py implements functions to gather datatype constraints from RDF triples and functions to apply the constraints to validate strings.

The **Datatype_Constraints** Module defines an “abstract” **DatatypeConstraints** class and a subclass for each type of constraint. For example, the module defines the **VocabularyDatatypeConstraints** subclass which knows how to validate strings against a vocabulary list. The module also contains a function **get_datatype_constraints()**, which examines the RDF triples to determine what kind of constraint is being requested and returns an instance of the appropriate subclass of **DatatypeConstraints**.

Each subclass of **DatatypeConstraints** has its own **validate(value)** function, which returns a list of **ConstraintError** messages if **value** is not valid. Each subclass also has its own constructor which knows how to build itself from the triples that **get_datatype_constraints()** passes to it.

The initial release of the toolkit implements only one subclass, the **VocabularyDatatypeConstraints** subclass. The procedure for adding another subclass is as follows:

- Add a new subclass of **DatatypeConstraints** that follows the API
  - It has a constructor that knows how to build itself from the triples.
  - It has a validator that returns a list of **ErrorMessage** objects.
  - It has a **describe()** function that returns a plain-text description of itself.

- Add a new case clause to **get_datatype_constraints()** to recognize the new constraint type and create an instance of the new subclass of **DatatypeConstraints**.

An additional feature in this module is the **DatatypeException**. This allows subclass constructors to communicate error conditions if they fail during object creation by raising a **DatatypeException** containing a list of **ErrorMessage** objects (e.g. **raise DatatypeException(errmsgs)**. The function creating the object can catch the exception as **exc** and recover the list of **ErrorMessages** as **exc**.**error_messages**. Although the Vocabulary subclass does not need this feature, future subclasses may.

5.7 Class_Constraints

The **class_constraints.py** module implements the **ClassConstraints** class. Each **ClassConstraints** object contains a set of **PropertyConstraints** objects, each of which describes the cardinality and/or range constraints for a single property.

This module's main feature is its **get_class_constraints()** function, which builds a **ClassConstraints** object and its constituent **PropertyConstraints** objects from RDF triples. The Ontology module invokes this function to build a **ClassConstraints** object for each class in the ontology that has **OWL.class** constraints. The remainder of this section describes this function.

The **get_class_constraints()** function accepts ***ontospy's*** list of RDF triples for a single ontology class and sorts them by subject, predicate and object into a ***spo dictionary***, ***spo[subject][predicate][object]***, as described in section 5.1. One subject is the parent node (the single ontology class to which these triples apply) and the rest of the subjects are intermediate nodes, or ***BNodes***. The ***BNodes*** contain the constraint information.

Consider the dictionary ***spo[bnode]*** for one of the ***BNodes***. This dictionary maps several predicates to their value, and each predicate should have one and only one value. The predicates and their value are as follows.

Predicate	Value	Comment
RDF.type	**OWL.Restriction** or something else	The toolkit supports only **OWL.Restrictions**
OWL.onProperty	The URI of the property being constrained.	The toolkit requires this predicate to be present
OWL.onDataRange	The property's range (allowed value type)	Range is a **datatype**
OWL.onClass	The property's range (allowed value type)	Range is an ontology class
OWL.minCardinality	Minimum number of values allowed for this property	Range constraint may not be specified.
OWL.maxCardinality	Maximum number of values allowed for this property	Range constraint may not be specified.
OWL.cardinality	Exact number of values required for this property	Range constraint may not be specified.
OWL.minQualifiedCardinality	Minimum number of values allowed for this property	Range constraint must also be specified.
OWL.maxQualifiedCardinality	Maximum number of values allowed for this property	Range constraint must also be specified.
OWL.qualifiedCardinality	Exact number of values required for this property	Range constraint must also be specified.
RDF.first, **RDF.rest**	Defines a linked list of **BNodes**.	The ***vocabulary*** datatype uses this to specify valid strings.

Through these predicates, each ***bnode*** specifies a property and its cardinality and/or range constraints. For each ***bnode***, **get_class_constraints()** collects and encapsulates the constraints into a **PropertyConstraints** object and stores the object in the **ClassConstraints** object. The function also identifies and reports inconsistencies amongst the constraints.

5.8 Property_Constraints

The **property_constraints.py** module implements the **PropertyConstraints** class. Each instance of this class describes the cardinality and/or range constraints for a single property.

The **PropertyConstraints** class contains (too) many lines of code dedicated to checking for inconsistencies. However, the class's main feature is its **merge_parent()** function, which handles constraint inheritance.

A **PropertyConstraints** object's **merge_parent()** function accepts the **PropertyConstraints** object from which to inherit constraints and updates its own constraints to contain the result of the inheritance. Section 5.3.2 explicitly tabulates all possible constraint inheritance situations and their results.

5.9 The XSDValidator Module

The XSD Validator Module implements a class that uses a python's built-in XML Schema Definition (***XSD***) schema validator to validate simple **datatype** objects such as integers, dates and strings.

The **xsd_validator.py** module implements the function **validate_xsd()**, which returns an **ErrorMessage** if a specified string is not valid for a specified **xsd** type. It relies on python's **XMLSchema** class, whose **assertValid()** method validates an input string against an XML schema.

The **validate_xsd()** function first fetches a cached **XMLSchema** object for the specified **xsd** type. If one has not been cached, it creates one with a simple schema for the specified type. For example, for an ***xsd:integer*** type, it creates an **XMLSchema** with this schema:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="value"  type="xsd:integer"/>
</xsd:schema>
```

Then it uses the specified string to create a simple XML document. For example, the XML document for the string "foo" would be this:

```
<value>foo</value>
```

Finally, **validate_xsd()** applies the simple XML document to the XMLSchema object to validate the string; if there are error messages, it wraps them in ErrorMessage objects.

5.10 Context and Namespaces

The context.py file provides useful functions to organize and access namespaces. The Validator Toolkit uses the following definitions and provides functions to convert between strings, uris and qnames.

5.10.1 Concepts and Strings

URI. A Universal Resource Indicator. URIs are unique; no two URIs refer to the same thing. A URI consists of two components: its **namespace** and its **name**.

URI string. A URI represented as a string, e.g. 'http://www.w3.org/2001/XMLSchema#integer'. It is the concatenation of **namespace string** and its **name string**.

Namespace. A location of a group of related URIs. Namespaces are unique; no two namespaces refer to the same location.

Namespace string. A namespace represented as a string, e.g. 'http://www.w3.org/2001/XMLSchema#'

Name. Something that identifies a property, class or object.

Name string. A name represented as a string, e.g. 'integer'.

5.10.2 Mappings and Abbreviations

Qualifier string. An abbreviation for a namespace string.

Binding. The association of a qualifier string with a namespace string.

N**amespace mapping.** A mapping that relates qualifier strings to namespace strings. For example, the namespace mapping may map the qualifier string 'xsd' to namespace string 'http://www.w3.org/2001/XMLSchema#'. The mapping may relate multiple qualifier strings to the same namespace string. ***A qualifier string is meaningless without a namespace mapping.***

There is no universal namespace mapping. Each ontology and data file defines its own, and may map different qualifier strings to the same namespace. Namespaces are absolute across ontologies. Qualifier strings are not.

I**nverse namespace mapping.** A mapping from namespace string to qualifier string derived by inverting a namespace mapping. If the original namespace mapping relates multiple qualifier strings to the same namespace string, the Validator Toolkit chooses one of them arbitrarily for the inverse mapping.

Qualified Name, or **QName**. An abbreviated form of a URI string: the concatenation of a qualifier string, a colon, and a name string. ***A QName is meaningless without a namespace mapping.***

5.10.3 Implementation in Python objects

URI Object. An rdflib.term.URIRef that wraps a URI string and offers some useful parsing and name-building functions. Unfortunately, when a data file uses qnames instead of full URIs, ontospy produces rdflib.term.URIRef objects wrapping the qname. ***These objects are meaningless***, but they can be normalized using a namespace mapping.

Namespace Object. A rdflib.term.Namespace object that wraps a namespace string and offers some useful parsing and name-building functions. The rdflib module contains predefined Namespace objects, including rdflib.RDF and rdflib.OWL.

5.11 Reports

The validator’s main product is reports. To ensure uniform message format, the **Message** module defines the **ErrorMessage** class and a few trivial subclasses (e.g., **OntologyError**, **ConstraintError**, etc.). These classes include optional message attributes, such as line number, message content, the offending class, the offending property, and the name of the function that produced the message. They also contain a rendering function the constructs error messages from the available attributes in a uniform way.

When the validator encounters an error, it creates an instance of one of these classes, sets the appropriate attributes, and stores the object until it is ready to report the errors. When it is ready to report the errors, the validator sorts the **ErrorMessage** objects by class, property or line number, removes duplicates, and calls their rendering function to get the message strings.

6 Known Issues

- **base64Binary.** When **ontospy** builds its triples and encounters an item of @type xsd:base64Binary, it converts base64-decodes the corresponding @value to yield the binary string. The validator does not try to validate xsd:base64Binary types.
- **Line numbers.** The validator sometimes gets the line number wrong. The reported line number is usually the line number of the nearest @type statement but is wrong or undefined when @type statements are missing.
- **BNode errors.** The validator does not support **olo:slots** and **olo:items.** If the data includes these, the validator produces meaningless reports citing errors in **BNodes**.

7 Future

The UCO/CASE Validation Toolkit supports OWL-based property constraint and does not support SHACL. Whereas there are already many products that validate SHACL-based constraints, this toolkit will probably not be upgraded to support SHACL-based constraints.

That said, a few enhancements may be added if requested. These might include but are not restricted to:

- Bug fixes and product maintenance.
- Corrections to misinterpretations of how ontologies and data work together.
- Support for additional **datatypes**, such as integer ranges. The current version supports only simple and Vocabulary datatypes.
- Allowing subclasses in datatype inheritance. The toolkit currently does not realize that a vocabulary datatype is a subclass of the string datatype, and therefore can be inherited.
- Support for a wider range of OWL predicates. The toolkit currently supports only the most-frequently used ones.
- Support for SHACL if requested.

Appendix A. Ontospy Resources

Ontospy is an open-source python program that ingests ontologies, decomposes them into RDF triples, organizes their classes properties and shapes into inheritance hierarchies, and offers an API that allows us to examine them. It can also ingest JSON-LD files and decompose them into RDF triples.

Some of **Ontospy**'s functions and attributes are listed below.

Ontospy

o = Ontospy(uri_or_path='Oresteia.json', rdf_format='jsonld'): ingest JSON-LD file.
o = Ontospy(uri_or_path='/path/to/turtle/directory'): ingest ontology turtle files.
o.all_classes: List of all the classes in the ontology.
o.all_properties: List of all the properties in the ontology.
o.all_properties_object: List of properties with an owl:ObjectProperty.
o.all_properties_datatype: List of all properties with an owl.DatatypeProperty.
o.namespaces: List of all the prefixes in the ontology and their corresponding URI.
o.rdflib_graph: The RDF triples in an rdflib.Graph object.

Class

c.uri: The class expressed as a URI.
c.qname: The Qname of the class, e.g., "types:Hash".
c.parents(): List of parent classes If P is the parent of C, then C has property rdfs:subClassOf P.
c.ancestors(): List of parent classes, grandparent classes, etc.
c.children(): List of classes that have c as a parent.
c.descendants(): List of children classes, grandchildren classes, etc.
c.triples: List of all RDF triples for this class.
c.domain_of: List of properties that this class may have.
c.domain_of_inferred: Dictionary of c and its ancestor classes mapped to lists of properties.
c.range_of: List of properties that may have an instance of this class as its value.
c.range_of_inferred: Dictionary of c and its ancestor classes mapped to lists of their ranges.
c.rdftype: Type of class: either RDFS:Datatype or OWL:Class.

Property

p.uri: The property expressed as a URI.
p.qname: The Qname of this property, e.g., "observable:hash".
p.domains: List of Classes that may have this property.
p.ranges: List of Classes that this property's value may belong to.
p.parents(): List of parent properties.
p.children(): List of child properties.
p.descendants(): List of descendant properties.
p.rdftype: Type of class: either owl:DatatypeProperty or owl:ObjectProperty.

Appendix B. Toolkit Modules

Primary Module	Description
describe.py	Main program to print metadata of serialized files.
serialize.py	Main program to serialize ontology and JSON-LD data files
validate.py	Main program to validate JSON-LD file against an ontology
v4_to_v5.py	Main program to convert data from UCO version 0.4 to UCO version 0.5

Additional Module	Description
casedata.py	Ingest and encapsulate JSON-LD file
class_constraints.py	Build property constraints for properties in a class from RDF triples
datatype_constraints.py	Validate strings against vocabulary
message.py	Encapsulate and provide uniform output of reports
context.py	Manage namespaces
ontology.py	Ingest and encapsulate turtle files; compute constraints and ontology info
precondition.py	Precondition and postcondition JSON-LD files
property_constraints.py	Assemble a property's constraints, inherit constraints from parent
serializer.py	Serialize and deserialize ontology and JSON-LD files
triples.py	Convert RDF triples to a ***spo*** dictionary
validator.py	Validate JSON-LD data files against turtle-based ontologies.
xsd_validator.py	Validate strings against simple datatypes (e.g. integer, date, float, etc.)