2023

Open Model

D2.4 CUDS classes and API

DOCUMENT CONTROL

Document Type	Technical Report
Status	Final
Version	1.0
Responsible	Adham Hashibon (UCL)
Author(s)	Francesca L. Bleken (SINTEF), Jesper Friis (SINTEF)
Release Date	2023-01-31

ABSTRACT

This deliverable reports on the development of the interoperability foundations for OpenModel, which are based on a set of common universal data structures and APIs. OpenModel combines and has contributed to several existing technologies for semantic interoperability developed in other EU projects and combines them to suite the need to the OpenModel OIP.

CHANGE HISTORY

Version	Date	Comment
0.1	2023-01-09	First Draft with setup o headers. Francesca Bleken
0.2	2023-01-29	Placed the sections into context. Wrote about OTEAPI and mappings. Jesper Friis
0.3	2023-01-30	Updated introduction, Adham Hashibon
0.4	2023-01-30	Cleaned up introduction, added abstract and description of SimPhoNy CUDS, Jesper Friis
0.5	2023-01-31	Review by coordinator

1.0	2023-01-31	Final

DISSEMINATION LEVEL		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

TABLE OF CONTENT

Document Control 1
Abstract 1
Change History 1
Dissemination level 2
Table of Content 3
List of Figures 4
1 Introduction 5
2 The OpenModel interoperability framework 6
2.1.1 SYMPHONY CUDS 7
2.1.2 DLite 8
2.1.3 Converting beween CUDS and DLite 11
Additional components 11
2.1.4 OTEAPI 11
2.1.5 Tripper 13
3 Acknowledgment 15

LIST OF FIGURES

Figure 1. Reuse interoperability technologies from earlier efforts in OpenModel. 5

Figure 2. The OpenModel technology stack. 6

Figure 3. The OpenModel interoperability framework. 7

Figure 4. Example of how SimPhoNy works with linked data about a city from three different sources: the city's traffic authority, a map from a city guide, and the university registry. Each data source is represented using a different colour and column. 8

Figure 5. The DLite data model. 9

Figure 6. An OTEAPI pipeline consisting of two partial pipelines. 12

Figure 7. Separation of concerns. 13

Figure 8. Illustration of how mappings can be used to semantically enhance existing metadata. 14

DELIVERABLE 2.4: CUDS CLASSES AND API

1 INTRODUCTION

This deliverable D2.4 reports on work done in Task 2.3 and describes the efforts exerted in OpenModel to adapt existing and emerging Common Universal Data Structures (CUDS) and APIs via EU semantic interoperability and Open Simulation Platform (OSP) technologies, originally outlined in the EMMC-CSA project and implemented in a range of projects, including SimPhoNy, SimDOME and OntoTrans (see Figure 1). In Task 2.3 these technologies have been further developed and extended to support needs by the OpenModel OIP. A holistic approach was taken in OpenModel to allow support for as large an EU innovation ecosystem as possible. OpenModel OIP have been developed and designed to be accessed with multiple interoperability frameworks including SimPhoNy CUDS[1], SOFT/DLite[2] and OTEAPI[3] as well as of AiiDA[4] data structures, and can be adapted to any existing standard using the newly developed in OpenModel, like building and execution of declarative workflows. In particular the results of the work performed in Task 2.3 facilitate the seamless standardised integration of in principle, and in practice, any third party-physics-based models, solvers, post-processors and databases.

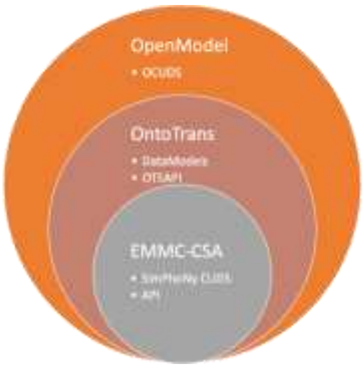


Figure 1. Reuse interoperability technologies from earlier efforts in OpenModel.

[1] GitHub - simphony/simphony-osp: A framework that aims to achieve interoperability between software such as simulation engines, databases and data repositories using a knowledge graph as the common language.
[2] https://github.com/SINTEF/dlite
[3] OTE-API Core (emmc-asbl.github.io) https://emmc-asbl.github.io/oteapi-core/latest/
[4] https://www.aiida.net/

Specifically, the following aspects have been targeted; (i) enhanced support for the Elementary Multi-perspective Materials Ontology (EMMO) and (ii) a universal plugin architecture for third party tools including support for web-API. Additionally, across the board enhancements to the overall performance of OpenModel OIP, especially for supporting declarative OpenModel workflows (ExecFlow and OntoFlow) has been achieved in collaboration with WP3.

We emphasise that the common data structures and APIs have been delivered in a manner that focuses on serving the existing and emerging OIP needs. The utilisation of logical reasoning for graph transversal in OntoFlow has been implemented in the tripper.mappings module and reported in deliverable D4.2. The remote access to HPC and other platforms (like Marketplaces) is handled via AiiDA and further described in deliverable D4.5. The planned javaScript frontend was not deemed necessary based on the design in Task 2.2. Instead we have focused on improving the Python interface of DLite (which is implemented in C), for easier integration with other Python-based technologies.

Figure 2 shows the semantic interoperability technology stack in OpenModel. To the left we have the generic technologies and to the right the components of the software ecosystem that implements them.

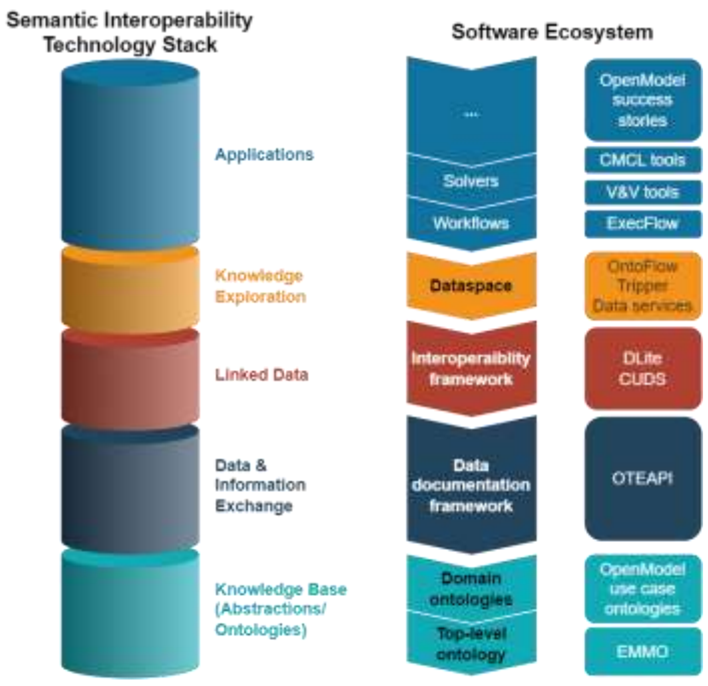

Semantic Interoperability Technology Stack
Applications
Knowledge Exploration
Linked Data
Data & Information Exchange
Knowledge Base (Abstractions/ Ontologies)
Software Ecosystem
...
Solvers
Workflows
Dataspace
Interoperaibility framework
Data documentation framework
Domain ontologies
Top-level ontology
OpenModel success stories
CMCL tools
V&V tools
ExecFlow
OntoFlow Tripper Data services
DLite CUDS
OTEAPI
OpenModel use case ontologies
EMMO

Figure 2. The OpenModel technology stack.

2 THE OPENMODEL INTEROPERABILITY FRAMEWORK

Figure 3 shows the OpenModel interoperability framework, which has been implemented in task 2.3. Its main responsibility is to provide a semantic data description and open APIs for accessing and working with data in a fully semantic way.

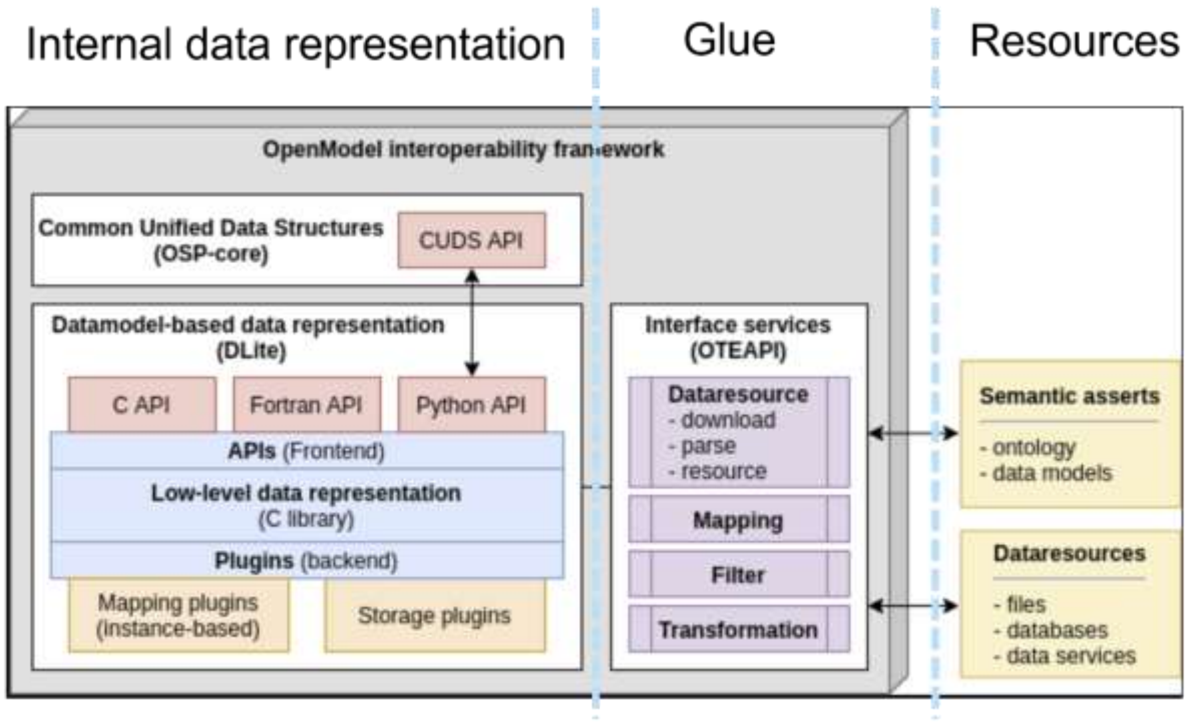

Internal data representation
Glue
Resources
OpenModel interoperability framework
Common Unified Data Structures (OSP-core)
CUDS API
Datamodel-based data representation (DLite)
C API
Fortran API
Python API
APIs (Frontend)
Low-level data representation (C library)
Plugins (backend)
Mapping plugins (instance-based)
Storage plugins
Interface services (OTEAPI)
Dataresource - download - parse - resource
Mapping
Filter
Transformation
Semantic asserts - ontology - data models
Dataresources - files - databases - data services

Figure 3. The OpenModel interoperability framework.

The core components of the OpenModel interoperability framework are further detailed in the following sub-sections.

2.1.1 SYMPHONY CUDS

The SimPhoNy OSP is a framework that aims to achieve interoperability between software such as simulation engines, databases and data repositories using a knowledge graph as the common language. It is focused on the domain of materials science.

SimPhoNy enables:

- Visualization and exploration of OWL ontologies and RDFS Vocabularies
- Wrappers: interfaces between ontologies and software products or digital objects
- Manipulation of ontology-based data: work with ontology individuals, transfer them among different software products using the wrappers, and query the knowledge graph

SimPhoNy is an ontology-based framework aimed at enabling interoperability between different simulation and data management tools (such as simulation engines, databases and data repositories) using a knowledge graph as the common language. It is focused on the domain of materials science.

Linked data is a format for structured data that facilitates the interoperability among different data sources. In particular, the data is structured as a directed graph, consistent of nodes and labeled arcs. With SimPhoNy, you can not only manipulate this linked data, but also transform existing non-linked data into linked data.

To better understand the idea of linked data, take a quick glance at the toy example below. It shows data about a city from three different data sources: the city's traffic authority, a map from a city guide, and the university registry. As some of the concepts are present in multiple datasets, the linked data representation naturally joins all of them into a single one.

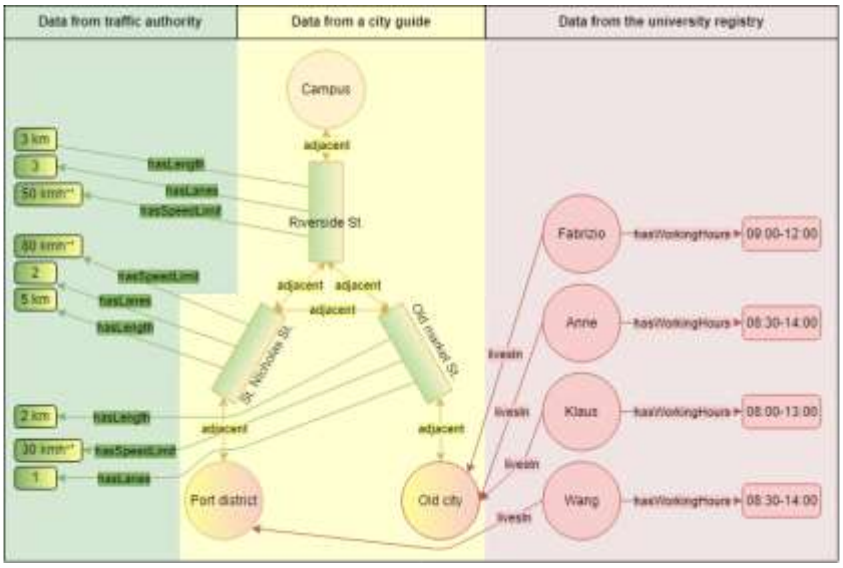


Figure 4. Example of how SimPhoNy works with linked data about a city from three different sources: the city's traffic authority, a map from a city guide, and the university registry. Each data source is represented using a different colour and column.

2.1.2 DLITE

In OpenModel, DLite is responsible for providing the underlying representation of the actual data. It is a software package designed for efficiently describing and working with scientific data, implemented in C and with both Fortran and Python bindings. Full semantic interoperability is obtained with mapping to ontologies (see Sections 2.1.4 and 2.1.5). DLite is pip installable from PyPi[5] as the package DLite-Python.

At the core of documenting the (scientific) data with DLite is the Entity. This is the metadata that the data provider needs to construct in order to describe the data.

[5] https://pypi.org/project/DLite-Python/

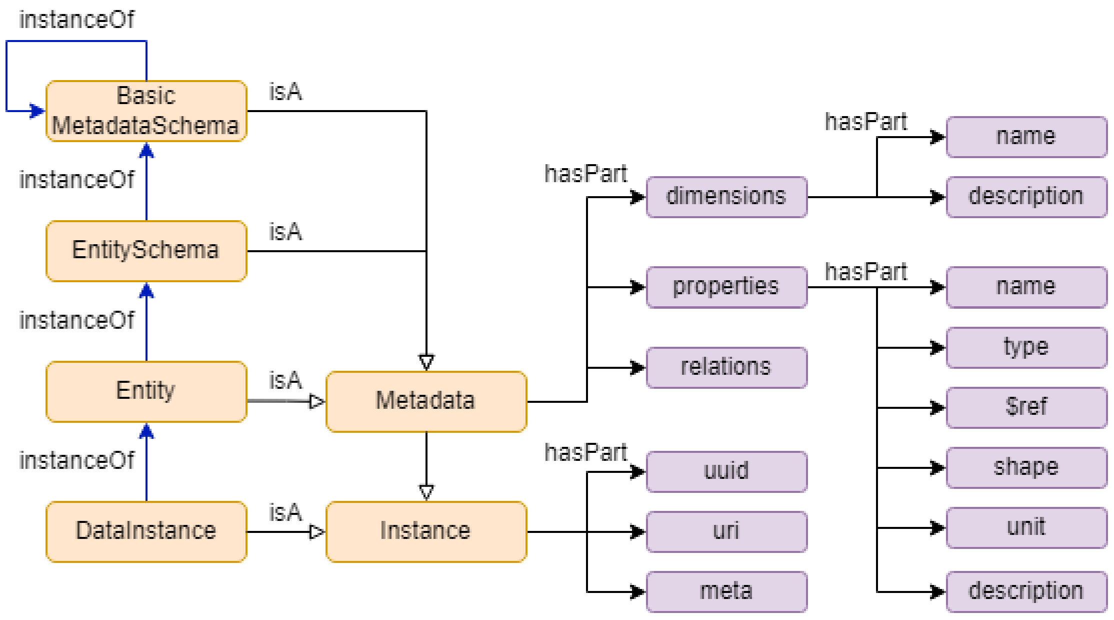


Figure 5. The DLite data model.

Figure 5 shows the DLite data model. Starting on the top left, the 'Basic MetadataSchema' is an instance of itself and is the most fundamental metadata schema of DLite. This is a metadata and defines the fundamental properties of the the DLite metadata seen in the top three left most lilla boxes and the right most boxes: dimensions, properties, realations, and their respective parts. The EntitySchema is a MetadataSchema and describes how an Entity is set up. An Entity is an EntitySchema that si specialized to document a specific data. A DataInstance is an Entity instantiated with data.

An example of and Entity that describes a molecule is shown below. Note that dimensions and properties that each instances of this molecule should have are described, but there is no data. This is the generic metadata for Molecule. It is worth mentioning that it is perfectly fine to make other Entities that describe a molecule, the one shown here is just an example. It is, however, important that the uris = namespace+name differ.

```
{
        "name": "Molecule",
        "version": "0.1",
        "namespace": "http://onto-ns.com/meta",
        "description": "A minimal description of a molecules",
        "dimensions": [
        {
        "name": "natoms",
        "description": "Number of atoms"
        },
        {
        "name": "ncoords",
        "description": "Number coordinates. Always 3"
        }
],
"properties": [
{
"name": "name",
"type": "string",
"description": "Name of the molecule."
},
{
"name": "positions",
"type": "double",
"dims": ["natoms", "ncoords"],
"unit": "Ångström",
"description": "Atomic positions in Cartesian coordinates."
},
{
"name": "symbols",
"type": "string",
"dims": ["natoms"],
"description": "Chemical symbols."
},
{
"name": "masses",
"type": "double",
"dims": ["natoms"],
"unit": "u",
"description": "Atomic masses."
},
{
"name": "groundstate_energy",
"type": "double",
"unit": "eV",
"description": "Molecule ground state energy."
}
]
}
```

While this is at first glance quite complex, this is designed so that an Entity should be relatively easy to construct for any type of Scientific Data.

DLite also provides some tools that can be used to work with the Entities and the Instances generated form the Entities (I.e.instances of entities populated with data).

Interoperability is achieved with mappings to an ontology (see below). This means that scientific data can be documented in two steps: 1. Map the data to a DLite metadata and 2. Map the DLite metadata to ontological concepts.

2.1.3 CONVERTING BEWEEN CUDS AND DLITE

CUDS can be considered equivalent to a knowledge graph which contains both ontologised concepts with relations and data. Thus, by default, CUDS do not have a clear structure. The data can be represented and grouped in various sets. Therefore, a generic converter between DLite datamodels (+ mappings) and CUDS is not conceptually feasible. The design of a converter is dependent on the user case and how the data will be manipulated. This means that targeted converters should be considered.

Building on a targeted converter specific to a given use case developed in OntoTrans, a bare-bone repository has been created, dlite-cuds. This application considers that a CUDS individual belongs to one class representing an object that has properties. For example, a component is made of material. The material information will correspond to a DLite Entity. But this component also has a geometrical description, and it would correspond to a different DLite Entity. In a CUDS, the same individual can belong to different classes and the various properties blended. The dlite-cuds converter will fail in this situation. In addition, all properties are expected with values and not as other class with properties. Therefore, in general, the CUDS content will need to be split to apply this converter.

The converter will then extract the structure and definition to generate automatically a DLite Entity and mappings. At this stage, the relation to be considered for identifying the properties must be specified in the configuration. This DLite Entity will be available to generate DLite instances, and also serve to create new CUDS.

The application of this converter requires the user to carefully consider the data structure to be mapped between DLite and CUDS representation. The user also needs to consider how the relation to other data must be kept toward the rest of the graph (or collection in the case of DLite) in order to avoid duplicates or loose data provenance.

Additional components

2.1.4 OTEAPI

The Ontology Translation Environment API (OTEAPI) is a framework for semantic data documentation, originally developed in OntoTrans. It is agnostic to the underlying interoperability system, but in OpenModel we are using it together with DLite (with its interfaces to CUDS).

It is used in OpenModel to document data sources and consumers. In ExecFlow it is used to provide semantic description of the input and output a model expects. The connection between AiiDA and OTEAPI pipelines are further described in deliverable D4.5.

OTEAPI utilizes the “pipe and filter” design pattern when implementing data documentation. A pipeline consists of a set of reusable filters connected by pipes. Each filter is configured separately. These configurations can be stored in a database and are the key for data documentation. There exists a set of different types of filters, but the most important are shown in Figure 6. A pipeline can be separated into two partial pipelines, where the first documents the data source and the second documents the data consumer. No data is transferred before the pipeline is executed.

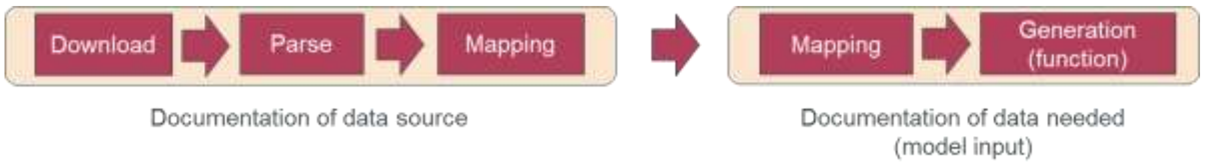

Download
Parse
Mapping
Mapping
Generation (function)
Documentation of data source
Documentation of data needed (model input)

Figure 6. An OTEAPI pipeline consisting of two partial pipelines.

A data source can be documented by a download, parse and mapping filter. The download filter knows how to access the data. The parse filter can parse the data and convert it to an internal representation. In OpenModel this is an instance of a data model. The mapping filter maps the data model to ontological concepts. Combined, they provide an interface for accessing the data source in a fully semantic way.

Similarly, a data consumer can be documented by combining a mapping and a generation filter. The mapping filter maps the expected data model to ontological concepts and the generation filter serialises the expected data model.

The mappings allow us to instantiate the expected data model from the data obtained from the data source (and possible other existing data). The implementation is based on the `tripper.mappings` module that is briefly described in the next section with DLite-specific functionality implemented in DLite.

An important aspect of OTEAPI is that it supports separation of concern as shown in Figure 7. In order for a data provider to make the data source semantically available, he/she only has to create a data model that represent the data and map the properties of the data model to ontological concepts. The mappings may be done in collaboration with an ontologist, especially if existing domain ontologies must be extended to capture the semantic meaning of the data. The model provider/data consumer has a similar task when semantically describing the model input. The data and model providers can work completely independent and may not even know about each other. The ontologist will, in addition to help the data and model providers, investigate the semantic connections between properties of the source and target data models. A dedicated software engineer can implement the low-level parser (that populates an instance of the data model from the data source), generator (that serialises an instance of the data model) and semantic conversion functions (for converting between source and target properties).

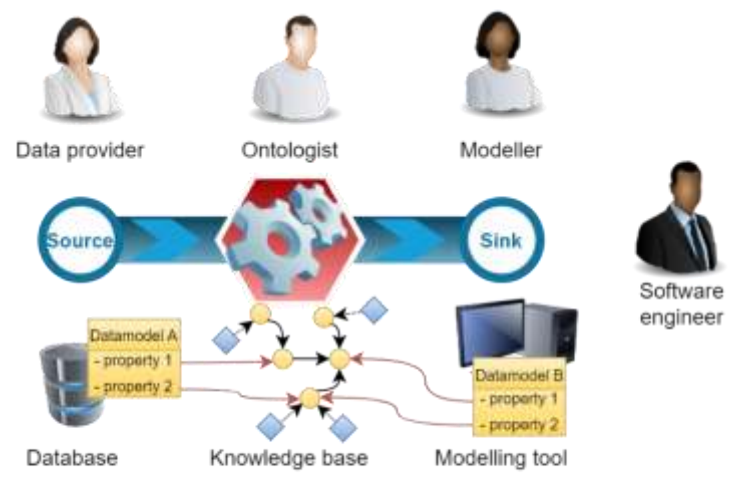


Figure 7. Separation of concerns.

More details can be found in the OTEAPI documentation: https://emmc-asbl.github.io/oteapi-core/latest/.

2.1.5 TRIPPER

Tripper[6] is a triplestore wrapper that provides a simple and consistent interface to a range of triplestore backends. Most components in OpenModel that need access to a triplestore, are now doing that through tripper. The benefit for OpenModel, is that we do not become tired to a given triplestore and that we easily can utilise and combine knowledge from multiple triplestores.

An important contribution to Tripper developed in OpenModel is the `tripper.mappings` module, which allows to traverse a knowledge base that describes a set of data sources and models and find possible routes to obtain a given piece of needed information by fetching from the data sources and invoking data models. This module is currently utilised in OntoFlow and further described in deliverable D4.2.

Currently, Tripper implements the following backends: DLite Collection, ontopy, rdflib and sparqlwrapper. ontopy (EMMOntoPy)is an extension of OwlReady2 and therefore uses the OwlReady2 triple store. While none of these backends are backends that will probably be used in production, they are good candidates for testing and pre-production, depending on the needs of the project. Tripper provides a simple and consistent interface to a range of triplestore backends. It strives for simplicity and is modelled after rdflib (with a few simplifications).

The following is taken from excerpts of the tripper tutorial. Please see the original documentation for more information.

To make it easy to work with IRIs, provide Tripper a set of pre-defined namespaces, like `XSD.float`. New namespaces can be defined with the `tripper.Namespace` class. A triplestore wrapper is created with the `tripper.Triplestore` class.

Different backends can be easily used by initializing the Triplestore object using the backend keyword. For example, to initialize with an rdflib backend one can run the following:

[6] https://github.com/EMMC-ASBL/tripper, https://emmc-asbl.github.io/tripper/latest/

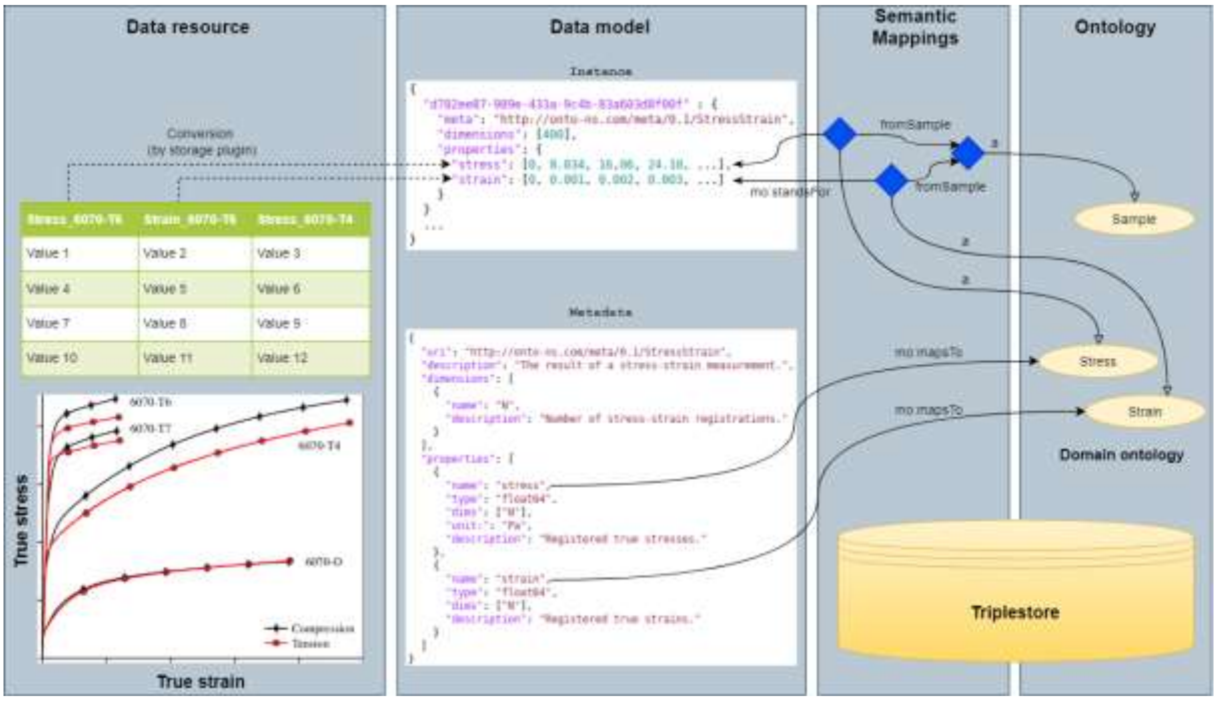
```
from tripper import Triplestore
ts = Triplestore(backend="rdflib")
```

Many namespaces come with the Triplestore module pre-defined. However, additional namespaces can be easily added with the `bind()` method. Namespace also supports access by label and IRI checking. New triples can be added either with the `parse()` method (for backends that support it) or the `add()` and `add_triples()` methods. For backends that support it the triplestore can be serialised using `serialize()`.

A set of convenient functions exists for simple queries, `including triples(), subjects(), predicates(), objects(), subject_predicates(), subject_objects(), predicate_objects()` and `value()`. Except for `value()`, they return the result as generators. The `query()` and `update()` methods can be used to query and update the triplestore using SPARQL.

Finally Triplestore has two specialized methods `add_mapsTo()` and `add_function()` that simplify working with mappings. `add_mapsTo()` is convinient for defining new mappings. It can also be used with DLite and SOFT7 data models. The add_function() describes a function and adds mappings for its arguments and return value(s).

Figure 8 shows how mappings to ontological concepts can be used to semantically enhance metadata. To the left we have a data source with experimental data of some stress-strain curves. In the middle bottom, we have a data model (metadata) that describes a single stress-strain curve. Above the metadata we see a json representation of an instance of this data model populated with data from the experiment. The datamodel is semantically enhanced by mapping its stress and strain properties to the ontological concepts 'Stress' and 'Strain', respectively. In the knowledge base we can now add individuals of these concepts that stands for the actual data properties. The raw data is not stored in the knowledge base, but is referred to from it.



Stress_6070-T6	Strain_6070-T6	Stress_6070-T4
Value 1	Value 2	Value 3
Value 4	Value 5	Value 6
Value 7	Value 8	Value 9
Value 10	Value 11	Value 12

```
{
  "d702ee87-989e-431a-9c4b-83a603d8f00f" : {
    "meta": "http://onto-ns.com/meta/0.1/StressStrain",
    "dimensions": [400],
    "properties": {
      "stress": [0, 8.034, 16.06, 24.10, ...],
      "strain": [0, 0.001, 0.002, 0.003, ...]
    }
  }
  ...
}
```

```
{
  "uri": "http://onto-ns.com/meta/0.1/StressStrain",
  "description": "The result of a stress-strain measurement.",
  "dimensions": [
    {
      "name": "N",
      "description": "Number of stress-strain registrations."
    }
  ],
  "properties": [
    {
      "name": "stress",
      "type": "float64",
      "dims": ["N"],
      "unit:": "Pa",
      "description": "Registered true stresses."
    },
    {
      "name": "strain",
      "type": "float64",
      "dims": ["N"],
      "description": "Registered true strains."
    }
  ]
}
```

Figure 8. Illustration of how mappings can be used to semantically enhance existing metadata.

3 ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 953167.