EMMC

EMMC-CSA

European Materials Modelling Council

Documentation

on

“Design and Implementation of metadata schema for syntactic and semantic interoperability”

TABLE OF CONTENT

The preparation of this report has received funding from the European Union‘s Horizon 2020 research and innovation programme under Grant Agreement No 723867

1. Executive summary

1.1 Description and objectives

The development of the European Materials & Modelling Ontology (EMMO) [1] has gained a lot of interest among the different stakeholders of EMMC, due to its great potential as an enabler for connecting scientific data and models in a semantic way. However, in order for people to start using it, tools and examples from real user cases are essential.

The objective of this document is to provide tools for working with EMMO and show a real user case on how EMMO may be used to enable vertical and horizontal interoperability and thereby enable people to start using EMMO. To support this, all code and examples have been made publicly available on GitHub under a permissive BSD[1] and MIT[2] licenses. Since long-term maintenance is essential for industrial adoption, an initial suggestion for a governance plan is also included.

This documentation contains three parts; updates of EMMO, tools for working with it, and a user case application demonstrating how EMMO can be used to achieve vertical and horizontal interoperability.

1.2 Major outcome

Updates of EMMO

EMMO has been released and is now available at https://github.com/emmo-repo. The documentation generated from the OWL file in the documentation of Materials Modelling Ontology in UML has also been updated and made publicly available.

Tools for working with EMMO

A reference API for working with EMMO has been developed and made available at https://github.com/emmo-repo/EMMO-python [2]. This API is in the form of a Python package called `emmo`. It is based on Owlready2[3] [3] and provides a natural way to represent EMMO in Python including read/modify/write OWL files as well as search and reasoning. This package also contains functionality for generating graphs for visualising EMMO and documenting it in various formats like pdf, html and markdown.

User case

To provide a realistic example of how EMMO can be used to achieve vertical and horizontal interoperability, a multiscale modelling user case was selected. The mechanical properties of a simple welded component consisting of an aluminium plate welded to a steel plate using an Al-based welding wire. In this process, a micrometre thick layer of intermetallic compounds is formed at the boundary between aluminium and steel. In order to describe the effect of this layer on the overall mechanical performance of the welded component, the modelling was performed at different granularity levels; at the component, microstructure and atomistic scale.

To enable **vertical interoperability**, a user case ontology (application ontology) was created by extending EMMO with all the classes and relations needed to describe this user case. A special emphasis was put on describing the properties transferred between the scales. The Python code defining and visualising this ontology

[1]The 3-Clause BSD License. Open Source Initiative: https://opensource.org/licenses/BSD-3-Clause.

[2]The MIT License. Open Source Initiative: https://opensource.org/licenses/MIT.

[3] Python package for ontology-oriented programming: https://bitbucket.org/jibalamy/owlready2/src/default/

is provided as a public example of how one can create an application specific ontology based on EMMO. It is available at https://github.com/emmo-repo/EMMO-python/tree/master/demo/vertical.

Horizontal interoperability is about using an ontological approach to connect different modelling types and codes for a single material user case, which is the key task on an interoperability environment. This demonstration tries to show a possible realisation of horizontal interoperability using the EMMO-based user case ontology created in the demonstration of vertical interoperability. The selected problem here, is to map an atomistic interface structure represented using ASE[4] (a Python package developed independently of EMMO) into our user case ontology, which here plays the role of an EMMO-based common representational system.

Since OWL is not well suited for representing complex materials data, like an atomistic structure, an open source metadata framework has been used to represent the actual data. The demo shows, in four steps, (i) how metadata can be generated from the ontology, (ii) how metadata can be defined for the independent application (ASE), (iii) instantiation of application data and (iv) mapping application data to an instance of metadata for the common representational system. The Python code implementing this demonstration can be found at https://github.com/emmo-repo/EMMO-python/tree/master/demo/vertical, while the metadata framework is available at https://github.com/jesper-friis/DLite.

2. Progress report (main activities)

2.1 Updates of EMMO

In preparation of the Second EC Workshop on Materials and Manufacturing Ontology (ECONTO2), EMMO was published on GitHub (https://github.com/EMMO-repo) alongside with a Python package for working with EMMO. The documentation has also been updated and made publicly available at https://github.com/emmo-repo/EMMO/blob/v0.9.9r2/doc/emmodoc.pdf.

A suggestion for a governance plan for EMMO is included in Annex 5.1.

2.2 Tools for working with EMMO

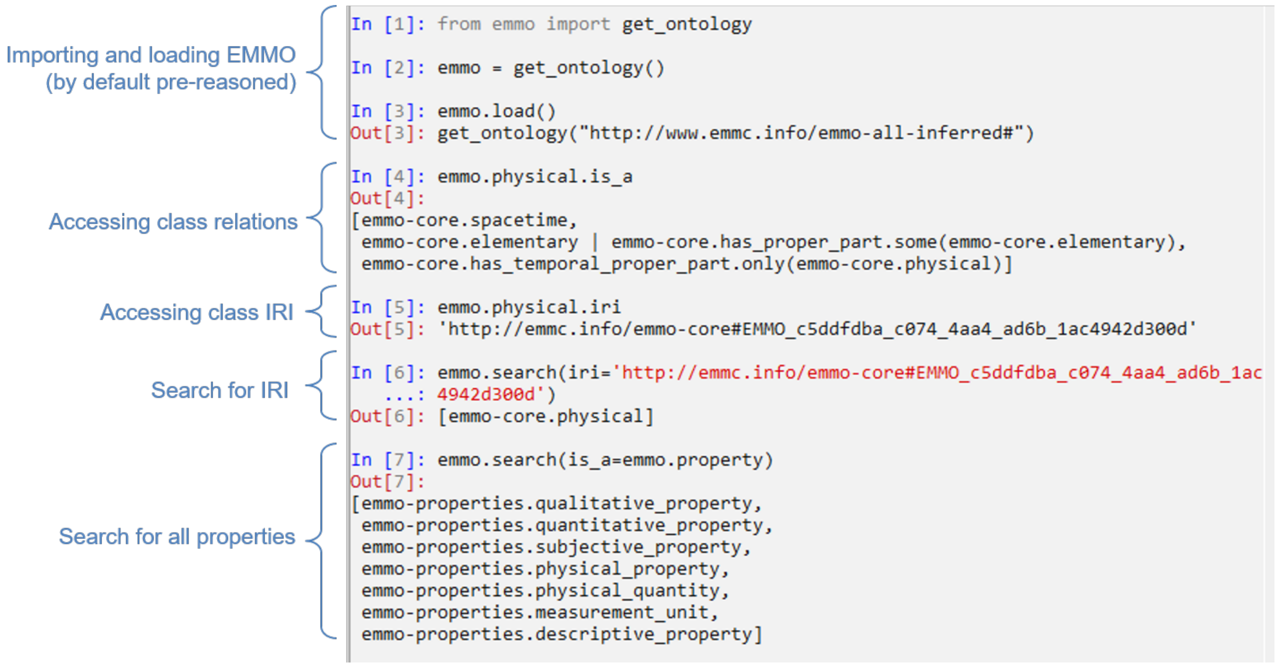
EMMO-python is a Python API for EMMO released under the Open source BSD license and available at https://github.com/emmo-repo/emmo-python/. It is based on Owlready2, which is a python package for ontology-oriented programming released under the GNU LGPL v3 License. Owlready2 may handle large ontologies (>10^9 classes), but care must be taken when reasoning in terms of RAM and time. Note that Owlready2 should not be imported in Python before EMMO-python. EMMO-python requires Python 3.5 or higher. Furthermore, the Python packages \`pydot\` and \`pandoc\` are required for generation of graphs and the EMMO-documentation, respectively. For reasoning, java is required.

While the EMMO itself is formulated using OWL, the EMMO-python package provides an easily available application for solving real problems, by providing a natural representation of it in Python. No knowledge of OWL is required to use emmo-python. Entities (OWL classes) are represented as Python classes, while OWL individuals are represented as instances of these. Relations are represented as class properties.

The workflow in EMMO-python typically starts with loading the EMMO ontology, preferably the previously inferred version available in the same repository. This is also the default if the ontology is not defined. Note that

[4]Atomistic Simulation Environment (ASE); https://wiki.fysik.dtu.dk/ase/.

ontologies must be in the rdfxml format. Figure 1 shows how to import the default ontology, 'emmo-all-inferred' and some examples on how to obtain information about the loaded ontology. For instance, you can access class relations, the class IRI (Internationalized Resourse Identifier), search for a given IRI or search for all properties.


Importing and loading EMMO (by default pre-reasoned)
Accessing class relations
Accessing class IRI
Search for IRI
Search for all properties

```
In [1]: from emmo import get_ontology

In [2]: emmo = get_ontology()

In [3]: emmo.load()
Out[3]: get_ontology("http://www.emmc.info/emmo-all-inferred#")

In [4]: emmo.physical.is_a
Out[4]:
[emmo-core.spacetime,
 emmo-core.elementary | emmo-core.has_proper_part.some(emmo-core.elementary),
 emmo-core.has_temporal_proper_part.only(emmo-core.physical)]

In [5]: emmo.physical.iri
Out[5]: 'http://emmc.info/emmo-core#EMMO_c5ddfdba_c074_4aa4_ad6b_1ac4942d300d'

In [6]: emmo.search(iri='http://emmc.info/emmo-core#EMMO_c5ddfdba_c074_4aa4_ad6b_1ac
   ...: 4942d300d')
Out[6]: [emmo-core.physical]

In [7]: emmo.search(is_a=emmo.property)
Out[7]:
[emmo-properties.qualitative_property,
 emmo-properties.quantitative_property,
 emmo-properties.subjective_property,
 emmo-properties.physical_property,
 emmo-properties.physical_quantity,
 emmo-properties.measurement_unit,
 emmo-properties.descriptive_property]
```

Figure 1. Examples of use of emmo-python.

The vertical and horizontal demo examples presented in this document show examples on how this package is used to expand EMMO in compliance with specific material research problems.

2.3 *User case application of EMMO*

This section shows a real application of how EMMO may be used to enable vertical and horizontal interoperability. The purpose is to provide an example and reusable patterns and scripts that will make it easier for the European materials modelling community to start using EMMO. To support this, all code and examples have been made publicly available on GitHub in the demo subfolder under the EMMO-python branch.

The actual work behind the user case is an ongoing effort within SFI Manufacturing[5]. Since the focus here is about interoperability, we have tried to simplify the user case as much as possible without losing the connection to the issues one meets in reality when performing multiscale modelling.

2.3.1 The user case

The selected user case is about multiscale modelling of welding aluminium to steel using an aluminium-based welding wire and how the thin layer of intermetallic compounds that are formed at the interface is influencing the overall strength and fracture properties of the component [4]. The goal of the multiscale modelling and the approach are schematically shown in Figure 2.

[5]A Norwegian centre for research-based innovation: https://www.sfimanufacturing.no/.

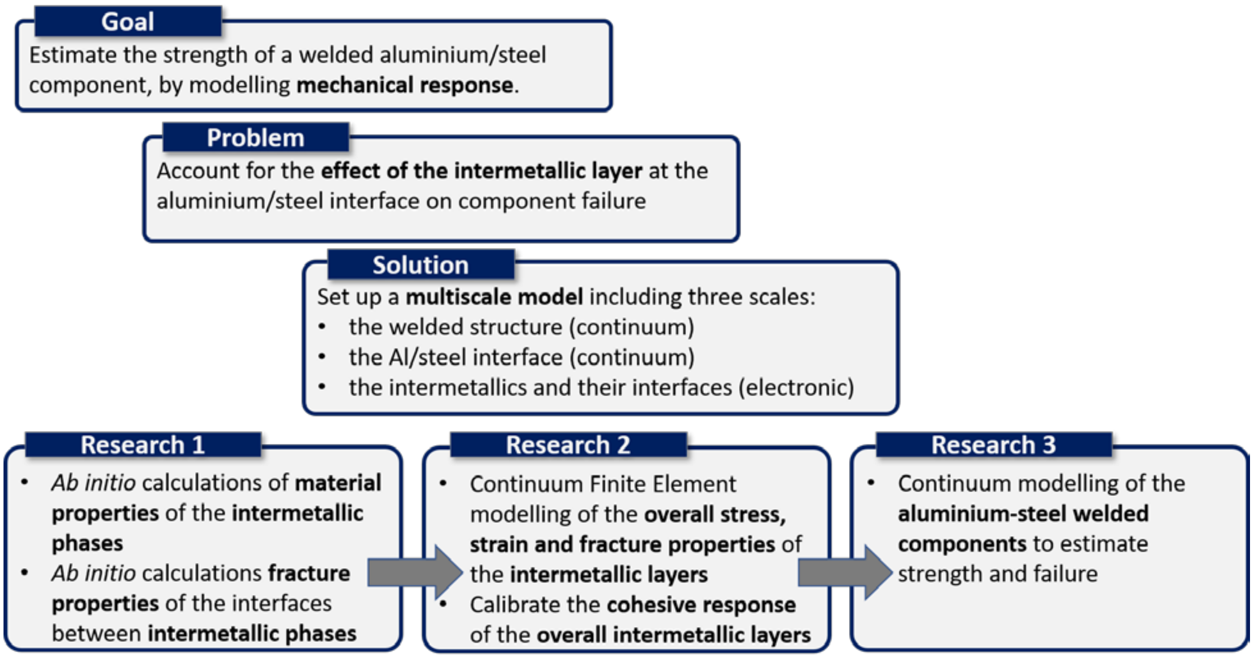

Goal
Estimate the strength of a welded aluminium/steel component, by modelling **mechanical response**.
Problem
Account for the **effect of the intermetallic layer** at the aluminium/steel interface on component failure
Solution
Set up a **multiscale model** including three scales:
• the welded structure (continuum)
• the Al/steel interface (continuum)
• the intermetallics and their interfaces (electronic)
Research 1
• *Ab initio* calculations of **material properties** of the **intermetallic phases**
• *Ab initio* calculations **fracture properties** of the interfaces between **intermetallic phases**
Research 2
• Continuum Finite Element modelling of the **overall stress, strain and fracture properties** of the **intermetallic layers**
• Calibrate the **cohesive response** of the **overall intermetallic layers**
Research 3
• Continuum modelling of the **aluminium-steel welded components** to estimate strength and failure

Figure 2. Schematic of the horizontal interoperability problem. Computational research at three different granularity levels is required. The arrows indicate data flow from one scale to another.

Figure 3 shows the three granularity levels considered in this user case. Since they are at different scales, we have chosen here to name them the component, microstructure and atomic scale.

Component scale

The component scale describes the overall mechanical behaviour of the welded structure and is the scale of interest for the end user. It predicts the strength of a welded structure (mechanical response and failure) under specific loading and is modelled using continuum structural mechanics based on Newtonian mechanics and solved using finite elements as implemented in the LS-DYNA software[6]. Constitutive materials relations for bulk aluminium and steel alloys are taken from the SIMLab Toolbox[7] materials model library. In order to be able to describe cracking at the aluminium/steel interface it is modelled using cohesive elements. However, since the materials model library currently does not include cohesive elements for the intermetallic layer, new elements were calibrated from a model of the interface at the microstructure scale.

Microstructure scale

A continuum representative volume element (RVE) for the microstructure scale was constructed (as shown in the middle of Figure 3) based on transmission electron microscopy (TEM) characterisation using scanning precision electron diffraction (SPED) [4]. The following sequence of intermetallic phases

$$\mathrm{Al} \mid \alpha\text{-}\mathrm{AlFeSi} \mid \theta\text{-}\mathrm{Fe_4Al_{13}} \mid \eta\text{-}\mathrm{Fe_2Al_5} \mid \mathrm{Fe}$$

[6] LS-DYNA. Livermore Software Technology Corporation (LSTC): http://www.lstc.com/products/ls-dyna.

[7] SIMLab Toolbox. Developed by the Norwegian centres for research-based innovation SFI SIMLab and SFI CASA: http://sfi-casa.no/contributing-to-the-simlab-toolbox/.

was observed when going from aluminium to the steel and used when setting up the RVE. This is consistent with phase stability when increasing the Fe-concentration in the binary Al-Fe phase diagram. Elastic constants of the aluminium and iron phases were taken from the SIMLab Toolbox, while the elastic constants for the intermetallic α-AlFeSi, θ-Fe_4Al_{13} and η-Fe_2Al_5 phases was calculated from first principles. The interfaces between the different phases were modelled with cohesive elements and calibrated using traction separations obtained from first principles. From the RVE, the cohesive behaviour of the intermetallic layer was calculated

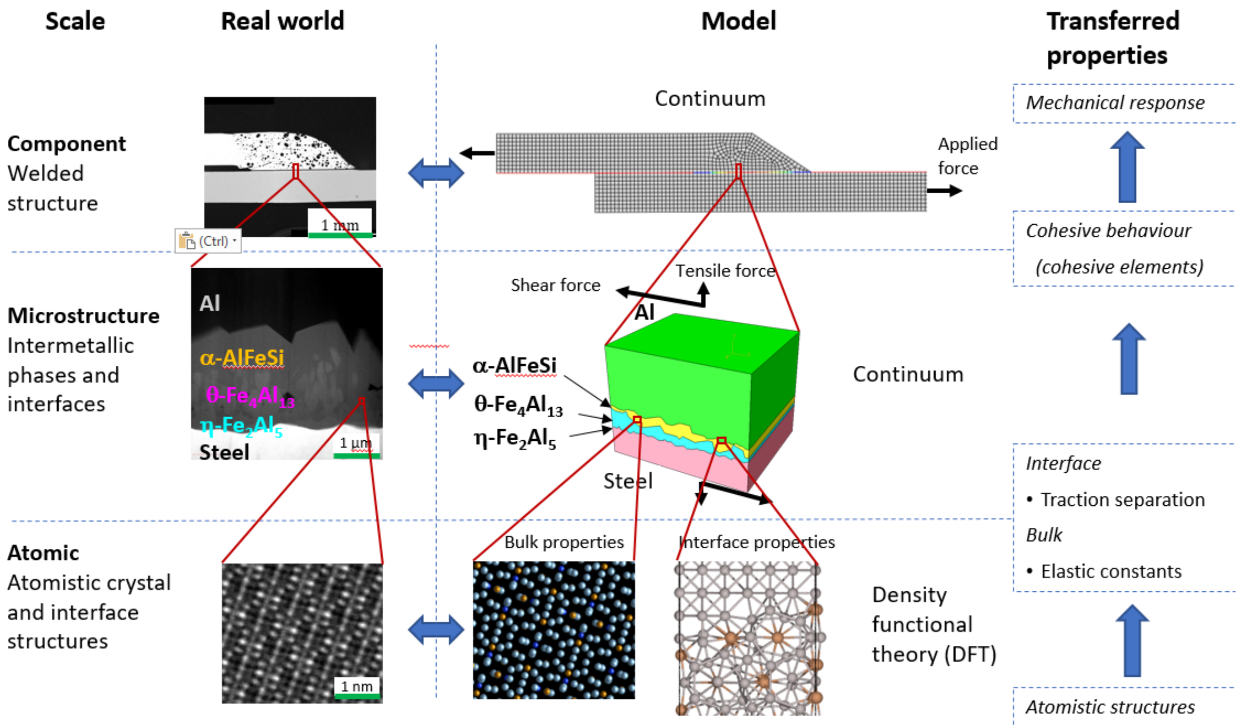


Figure 3. Schematic representation of vertical interoperability demonstrated on a welded Al-Fe structure. Experimental results at each level of granularity are shown to the left, with the corresponding computational

with LD-DYNA and used as input to the component scale.

Atomic scale

First bulk structures for the 5 phases (Al, α-AlFeSi, θ-Fe_4Al_{13}, η-Fe_2Al_5 and Fe) were set up and relaxed with density functional theory (DFT) using the Vienna ab-initio simulation package (VASP) [5]. From the bulk structures of the α-AlFeSi, θ-Fe_4Al_{13} and η-Fe_2Al_5 phases, elastic constants were calculated [6] and used as input to the microstructure scale. Based on the relaxed structures, minimum misfit interface structures for the four interfaces were set up and relaxed. Work of separation at the interfaces and within each phase were then obtained from a series of calculations by introducing a crack with increasing separation [7]. By postprocessing (curve fitting and differentiation) traction separations could then be obtained and used as input to calibrate the cohesive elements at the microstructure scale.

2.3.2 Vertical interoperability

Vertical Interoperability, is defined as *transfer of data between different codes and model types used to simulate the same material at two or more granularity levels (i.e. scales), through the ontological definition of the measurable properties of materials that will be stored within standard data structures and interchange format that will be defined during the project* [8]. The aim here is to show how EMMO can be used to achieve vertical interoperability. This is done by creating a user case ontology that extends EMMO with all concepts that are needed to describe the properties that are transferred between scales and how they are related to the materials entities for the different granularity levels.

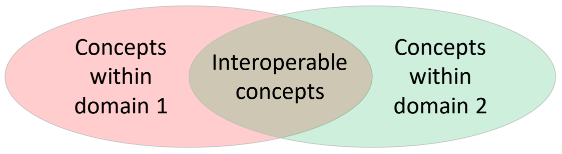


Figure 4. Only concepts in the intersection between two domains (i.e. are defined and has the same meaning within both domains) may be interoperable. This is especially important for vertical interoperability, where concepts named equally may have slightly different meaning at different granularity levels.

Figure 4 shows an important point. When connecting two domains, only concepts that are shared between the two domains and have the same meaning in both domains may be exchanged in a meaningful way. This is an

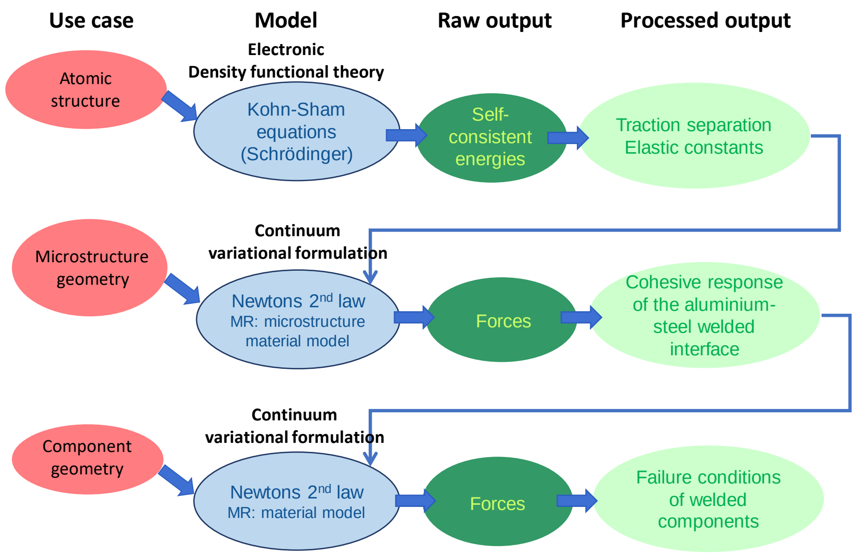

Use case
Model
Raw output
Processed output
Electronic Density functional theory
Atomic structure
Kohn-Sham equations (Schrödinger)
Self-consistent energies
Traction separation Elastic constants
Continuum variational formulation
Microstructure geometry
Newtons 2nd law MR: microstructure material model
Forces
Cohesive response of the aluminium-steel welded interface
Continuum variational formulation
Component geometry
Newtons 2nd law MR: material model
Forces
Failure conditions of welded components

Figure 5. MODA workflow for the user case.

important issue for vertical interoperability, where concepts named equally may have slightly different meaning at different granularity level. A common ontology that uniquely defines the transferred properties and relate them to materials entities at the different granularity levels is a key to handle this issue in a robust manner.

Figure 5 shows a MODA workflow for the user case. The codes used to solve the physical equations (VASP and LS-DYNA in this case) also provide internal workflows and postprocessing possibilities. For this user case, these details are not important. Here we only focus on the properties transferred between scales that are obtained by running the solver followed by postprocessing using facilities both within the solver codes and auxiliary scripts.

2.3.3 Defining the user case ontology

As mentioned above, a common ontology that uniquely defines the transferred properties and relates them to materials entities is a key for vertical interoperability. Hence, that is what we will focus on in our user case ontology. In Table 1 the materials entities that are modelled at each scale are listed together with the properties that are transferred between the scales. In the user case ontology, the materials entities at the atomic, microstructure and component scales are represented by the classes *crystal_unit_cell*[8], *RVE* and *welded_component*, respectively. The transferred properties are all properties of these classes or one of their parts and include:

- *lattice_vector*, *atomic_number* and *position* are signs that stands for the properties of *crystal_unit_cell* and its direct part *e-bonded_atom*. Together they describe an atomic structure, which is the material entity at the atomic scale.

Table 1. The material entities that are modelled at each scale (granularity level) and the corresponding input and output parameters that are transferred between the scales. The indentation of the material entities corresponds to direct parthood relations.

Scale	In/out	Material entity	Transferred properties
atomic	input	crystal - crystal_unit_cell - e-bonded_atom	 lattice_vector atomic_number, position
	output	- crystal_unit_cell - interface	stiffness_tensor traction_separation
microstructure	input	RVE - phase - boundary - interface	 stiffness_tensor traction_separation
	output	- boundary - interface	 traction_separation
component	input	welded_component - boundary - interface	 traction_separation
	output	welded_component	load_curve

- *Stiffness_tensor* is an objective quantitative property, i.e. it is defined by a physical law (in this case Hooke's law) and can be determined quantitatively with reference to this law by a measurement process or a simulation using a suitable model. It is a property of both *crystal_unit_cell* and a *phase* (which is a spatial direct part of *RVE*). Hence, *stiffness_tensor* is well-defined at both the atomic and microstructure scales.
- *traction_separation* is an objective quantitative property defined by a cohesive law describing the stress (force per interface area) required to initiate a crack (in a crystal, RVE or welded component) by separating the two sides a certain distance. At continuum level (microstructure and component scales), the traction separation is often described with a bilinear cohesive law relating the stress τ to the separation distance u via three materials parameters; initial stiffness K, damage onset separation u_0 and failure separation u_f:

[8]*atomic_unit_cell* might have been a better name for a general entity representing an atomic structure, but since we only deal with crystalline materials (and using a periodic dft code) in this user case, the term *crystal_unit_cell* was chosen, to emphasise that it is a direct part of a *crystal*.

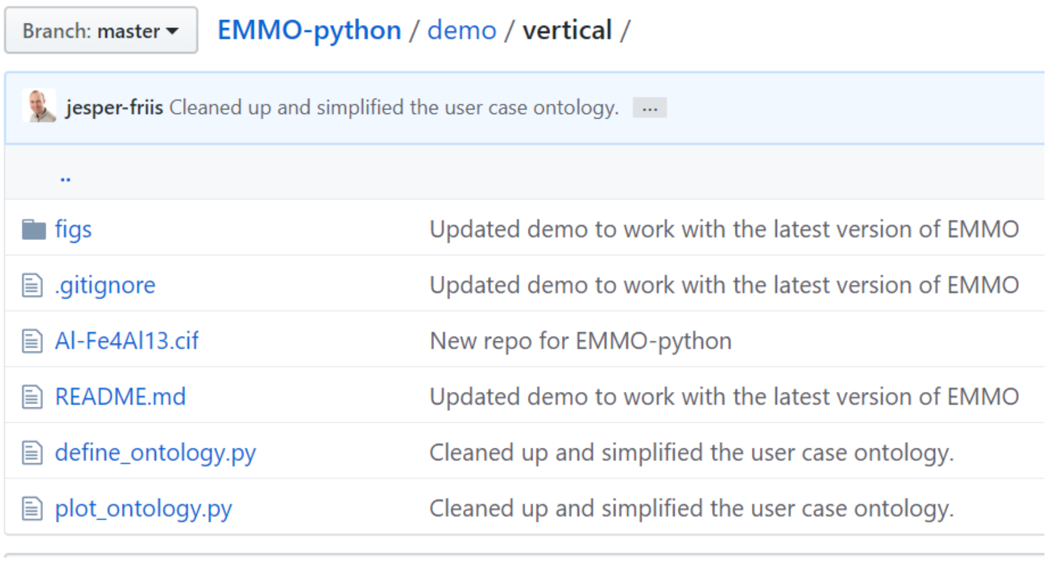
$$\tau(u) = \begin{cases} Ku, & u \leq u_0 \\ K(1-d)u, & u > u_0 \end{cases},$$

where $d = d(u) = u_f/u \times (u - u_0)/(u_f - u_0)$ is a damage variable ranging from 0 (no damage) to 1 (failure). Hence, material parameters K, u_0 and u_f would have been an alternative (and less general) way to specify the traction separation. At atomic level it is defined by the force (calculated from the spatial derivative of the energy) required to split a crystal structure a given distance.

- *load_curve* is a measure of the deformation of a material as a function of the applied force which is the property of interest for the end user.

Implementation

We are now ready to define our user case ontology by extending EMMO based on the materials entities and properties listed in in Table 1. Typically, this would be done using a tool like Protégé[9], but here we have chosen to define the ontology directly in Python using our Python API. The code is available on GitHub (Figure 6) and listed in Annex 5.3.This turns out to be a rather readable textual representation of the ontology (much more readable than standard OWL-DL formats like rdf/xml), which easily can be serialised into OWL-DL and loaded


Branch: master ▾ EMMO-python / demo / vertical /

jesper-friis Cleaned up and simplified the user case ontology. ...

..

figs	Updated demo to work with the latest version of EMMO
.gitignore	Updated demo to work with the latest version of EMMO
Al-Fe4Al13.cif	New repo for EMMO-python
README.md	Updated demo to work with the latest version of EMMO
define_ontology.py	Cleaned up and simplified the user case ontology.
plot_ontology.py	Cleaned up and simplified the user case ontology.

Figure 6. The code on GitHub for demonstrating vertical interoperability. The script define_ontology.py defines the user case ontology while plot_ontology.py plots it.

into Protégé.

Since the aim is to not just define the properties that are transferred between scales, but also to do it in practice, we need to add semantics for units and types to our ontology. Hence, we add two new relations

`has_unit` (subclass of `has_part`) and

`has_type` (subclass of `has_convention`)

together with corresponding inverse relations, `is_unit_for` and `is_type_of`, respectively. Annex 5.3, Figure 16, shows all the new classes in the user case ontology that are not already in EMMO. For simplicity models and properties used for postprocessing, etc. are not included in the ontology. Figure 17 in Annex 5.3 shows the material entities and their related properties.

[9]Protégé is a free, open-source ontology editor: https://protege.stanford.edu/.

2.3.4 Horizontal interoperability

Horizontal Interoperability is in deliverable D2.4 defined as *the use of different modelling types and codes for a single material user case (one user case, multiple models), that is pursued by an ontological approach to models and materials*. It allows to semantically map data between application-specific representations and a *common representational system*, which e.g. is paramount for an open simulation platform. This again opens the opportunity for providing the user with the choice of more than one software to address a single user case.

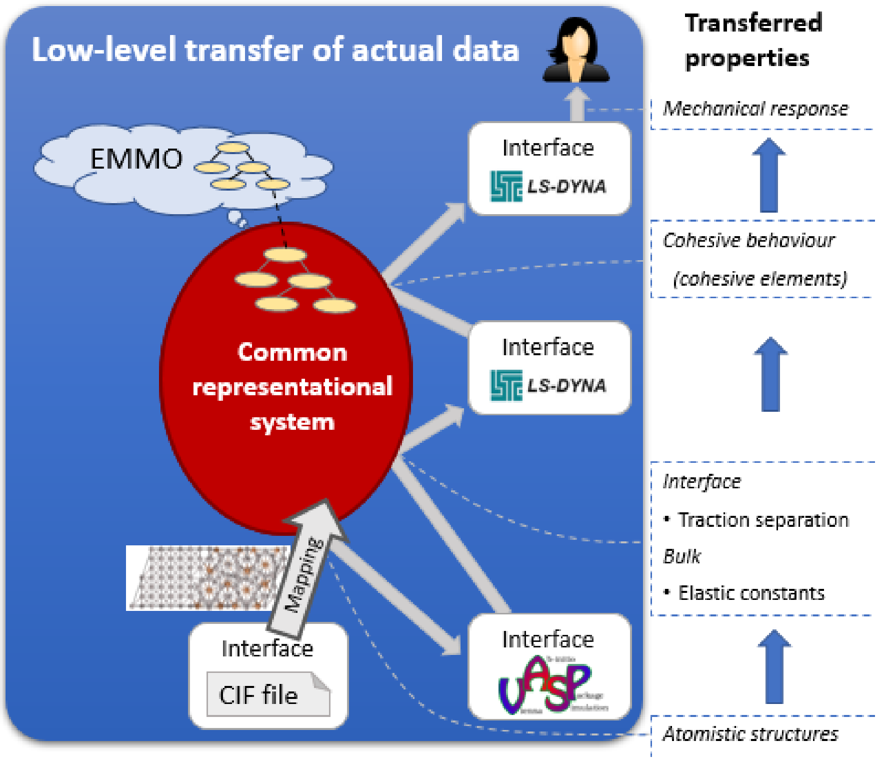


Figure 7. Demonstration of horizontal interoperability by using the user case ontology as a common representational system to allow mapping data between application-specific and common representations. The broken gray arrows represent semantically the same information mapped between three different representations. Here we will just look at the first mapping: mapping an atomic structure from a CIF file to our common representational system.

The basic idea here, is that we use our user case ontology defined in section 2.3.3 as our common representational system in Figure 7. With units and types related to the properties, it provides all semantics needed to allow mapping data between application representations and the common representational system. Since the concept is the same for all mappings, we have here only implemented the first mapping: mapping an atomic structure from a CIF file to our common representational system.

Since it is not practical to represent complex scientific data, like an atomic structure, as individuals with data properties in OWL-DL, we will here use a metadata framework for the low-level representation of the data. There are many metadata frameworks, like CUDS or FMI[10] that could be used. Here we have chosen a framework called DLite, which is described in more details in the box to the right. In the text that follows we will refer to SOFT for the ideas and DLite for the actual implementation used here. An important principle for a robust framework is a single source of information. This means that if we want to use a metadata framework, the

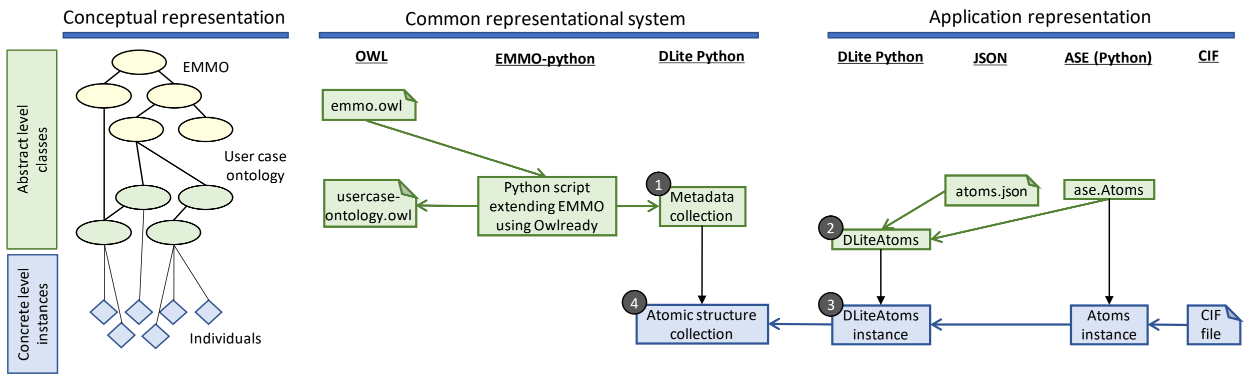


Figure 8. Overview of the different representations in the current user case. Along the vertical axis we have the level of generality, while different representations are shown along the horizontal axis, categorised into conceptual representation, common representational system and application representation. The latter two are subdivided into concrete representations. The classes/instances that are the results of each of the four steps are marked.

metadata should be generated automatically from the ontology to allow the ontology to be updated without worrying about creating inconsistencies with the metadata framework. Using a metadata framework, horizontal interoperability can be realised by the following four steps:

1. generate metadata from the common ontology
2. define application metadata
3. instantiate application data
4. map application data to instance of the common metadata

The results of these steps are shown in Figure 8 and related to the different representations and abstraction levels in the current user case. These four steps are conceptually rather generic when using a metadata framework. If one, on the other hand represents data as individuals with data properties directly in the ontology, step 1 is not needed, step 2 becomes "define application ontology" and step 4 "map application data to individuals of the common ontology".

[10]Industrial standard for high-performance wrapping between applications. See https://fmi-standard.org/.

Implementation

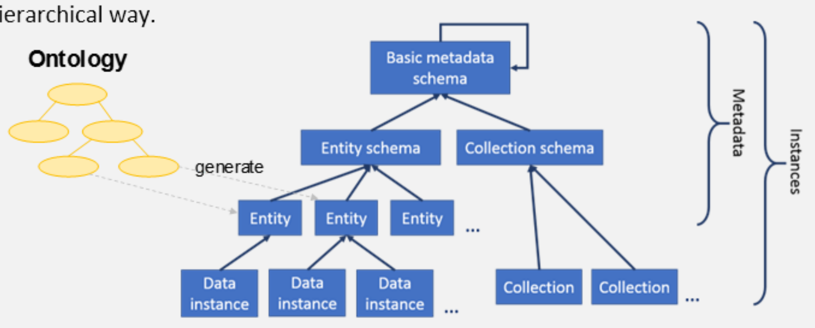
As shown in Figure 9, an implementation of these four steps are provided as an example made publicly available on GitHub. It is organised as one script for each of the four steps. In addition, there is a generic and reusable module emmo2meta, that does the actual work in step 1 and a JSON file used in step 2. Here follows a short comment of each of the four steps.

Step 1: Generate metadata from user case ontology
Implementation:
step1_generate_metadata.py
This script loads the user case ontology, then uses the emmo2meta module to generate a SOFT representation of the user case ontology and finally writes this representation to the JSON file `usercase_metadata.json`. The interesting part here is what emmo2meta does.

DLite – a light-weight data-centric metadata framework

Developer:	SINTEF
Operating system:	cross platform (Linux, Windows)
Language:	C, Python
Type:	data-centric metadata framework
License:	MIT
Web page:	currently https://github.com/jesper-friis/DLite

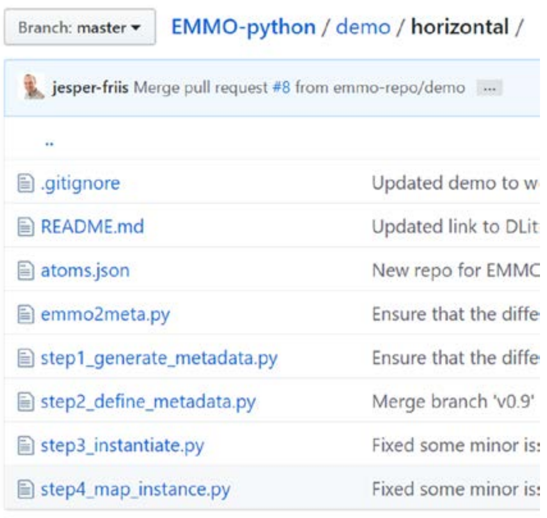
DLite is an open source C implementation of SINTEF Open Framework and Tools (SOFT). It provides a light-weight cross-platform C library with bindings to Python and Fortran, for working with and sharing scientific data in an interoperable way. As illustrated in the figure below, SOFT provides a way to describe data instances with formalised metadata, which in turn are described by their meta-metadata in a hierarchical way.



A SOFT entity describes a self-standing piece of information. It is possible to group several entities together in a collection and add relations between them. These relations belong to the context of the given collection and will not follow an entity if is added to another collection.
This feature makes it possible to represent an ontology using a collection. The benefit of this is to combine the application level semantics provided by the ontology with a simple and effective low-level data representation at code level as well as serialisation to different file formats or database.

From Figure 10 one sees that a SOFT entity has a header, a set of dimensions and a set of properties. This structure is recognised when generating the metadata, leading to the following mapping of the user case ontology:

- all OWL classes are mapped to SOFT entities, except for properties, types and units, that are treated especially as shown for a materials entity in Figure 11.
- all OWL relations are mapped into SOFT relations in the resulting collection.
- all restrictions (except for *has_property*, *has_unit* and *has_type* that are handled especially) are mapped into a SOFT instance of a Relation entity



Branch: master ▾ EMMO-python / demo / horizontal /

jesper-friis Merge pull request #8 from emmo-repo/demo ...

..

.gitignore	Updated demo to w
README.md	Updated link to DLit
atoms.json	New repo for EMMC
emmo2meta.py	Ensure that the diffe
step1_generate_metadata.py	Ensure that the diffe
step2_define_metadata.py	Merge branch 'v0.9'
step3_instantiate.py	Fixed some minor is:
step4_map_instance.py	Fixed some minor is:

Figure 9. The code on GitHub for demonstrating horizontal interoperability. Separate scripts are provided for the four steps.

e-bonded_atom (metadata)	
UUID	87134b67-9a0b-5a80-8380-74b3e74e2c2b
URI	http://emmc.info/emmo/demo/0.1/e_bonded_atom
Meta	http://meta.sintef.no/0.3/EntitySchema
Description	An electronic bonded atom that shares at least one electron to the atom_based entity of which is part of.

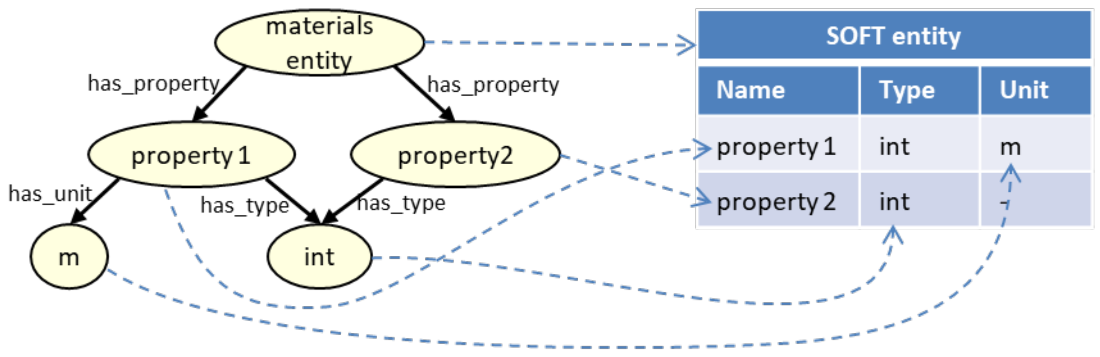
Dimensions	
Name	Description
ncoords	Number of coordinates (always 3)

Properties				
Name	Type	Dims	Unit	Description
atomic_number	int	-	-	The number of protons in the nucleus.
position	float	ncoords	m	Position of the atom.

Figure 10. SOFT entity for an e-bonded atom. It is uniquely identified by either its URI (consists of a namespace, version and name) or UUID (derived from the URI). Meta refers to the URI of the meta-metadata describing this entity and Description is a human description of the entity. It then has a set of named dimensions followed by a set of properties. Each property has a name, type, dimensionality, unit and a human description.

plus two relations in the resulting collection as shown in Figure 12(a). The reason for creating the additional Relation instance is to store the restriction type (*some*, *only*, *exactly*, *min* or *max*) and cardinality. A definition of the Relation entity is of course also included.

- all class constructs are mapped into a SOFT instance of a ClassConstruct entity plus a *has_argument* relation for each argument in the resulting collection as shown in Figure 12(b). Like for restrictions, the additional instance allows to specify the type of the class construct (*and*, *or*, *not*).



SOFT entity		
Name	Type	Unit
property1	int	m
property2	int	-

Figure 11. A materials entity together with its related properties, units and types is mapped into a single SOFT entity.

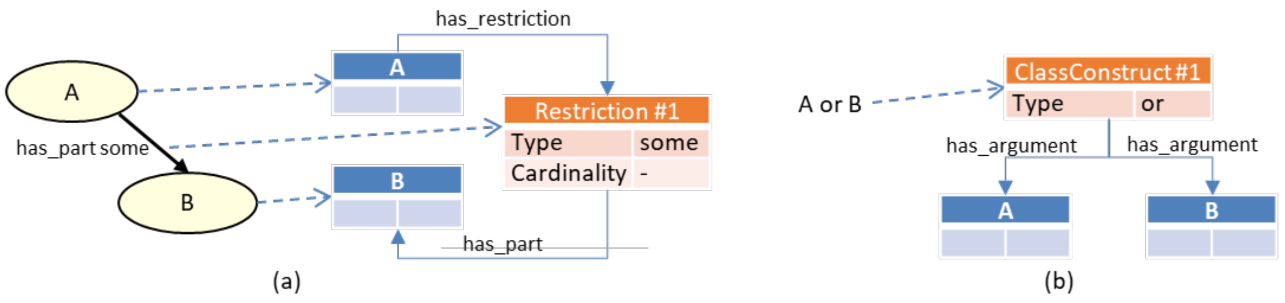


Figure 12. Mapping of (a) class restriction and (b) class construct. In both cases the mapping results in a SOFT data instance being created holding the extra information related to the restriction (type, cardinality) and class construct (type). The dashed blue arrows correspond to mappings while the solid blue arrows correspond to relations stored between SOFT entities stored in a collection.

Step 2: Define application metadata
Implementation: step2_define_metadata.py
This step depends on the application. In this case, we want to define SOFT metadata for an atomic structure represented in a CIF file. However, since ASE[4] already can read CIF files and represent them in a much simpler Atoms Python class, we define SOFT metadata for the Atoms class instead. The Python bindings for DLite provides a classfactory() function which makes this task very easy. As shown in the top part of Figure 13, we first create a JSON file atoms.json that defines an entity corresponding to the attributes and properties of the ASE Atoms class. We then use the classfactory() function to create a new Python class, *BaseAtoms,* that inherits from both the ASE Atoms class as well as from the entity we defined in JSON. In many cases this is sufficient. However, since the Atoms class has an `info` attribute which is of a type (dict) that doesn't correspond to any data type in DLite, we create a subclass *DLiteAtoms* of *BaseAtoms* and provide methods that converts this attribute back and forth between the ASE and DLite representations.

We can now replace any use of ASE Atoms class in existing code with the DLiteAtoms. The code will continue just like before, but it is now possible to ask the Atoms instances (aka objects) about their metadata or save it to any of the storage backends available in DLite. Loading a DLiteAtoms object from a storage backend is also possible. In short, what we have done is to add data-level semantics to the ASE Atoms class.

Step 3: Instantiate application data
Implementation: step3_instantiate.py
In this step we will use ASE to load an atomic Al/Fe_4Al_{13} interface structure from a CIF file and represent it as an instance of the metadata we defined in step 2, i.e. as a *DLiteAtoms* object. As shown in the bottom part of Figure 13, we first, use ASE to load the CIF file into an Atoms instance. Then we use the objectfactory() function provided by DLite to convert this instance into a DLiteAtoms object. Finally, we create a collection containing the new instance and its metadata (defined in the atoms.json file) and serialise it into a new JSON file; usercase_appdata.json.

Step 4: map application data to instance of the common metadata
Implementation: step4_map_instance.py
We have now defined SOFT metadata for the common representational system, i.e. our user case ontology (step 1) as well as for atomic structures (step 2). In step 3 we created a SOFT instance of an atomic Al/Fe_4Al_{13} interface structure. We are now ready to map this structure into SOFT instances of the user case ontology.

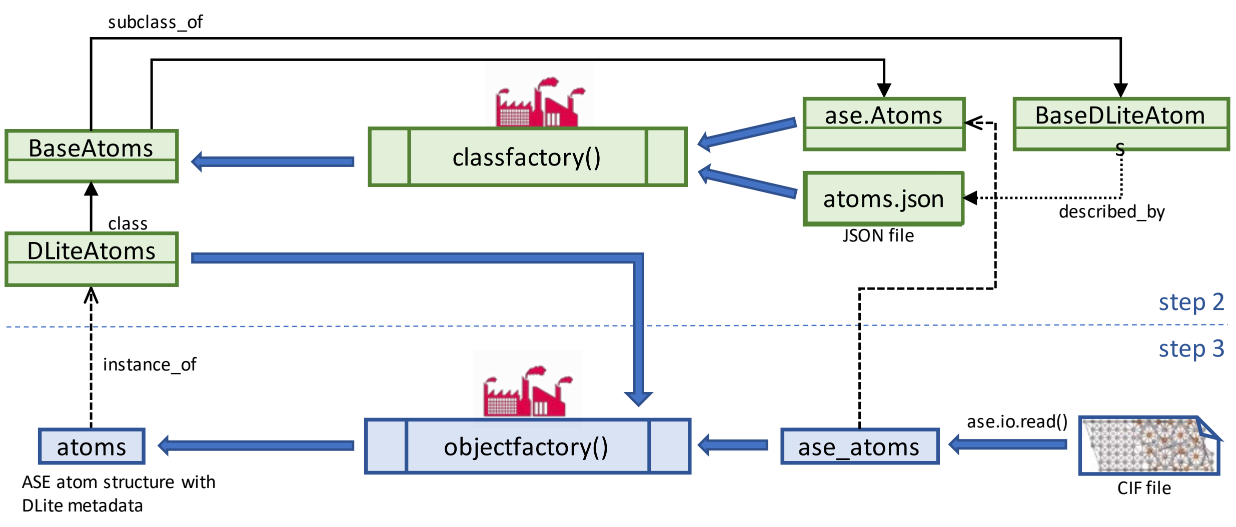
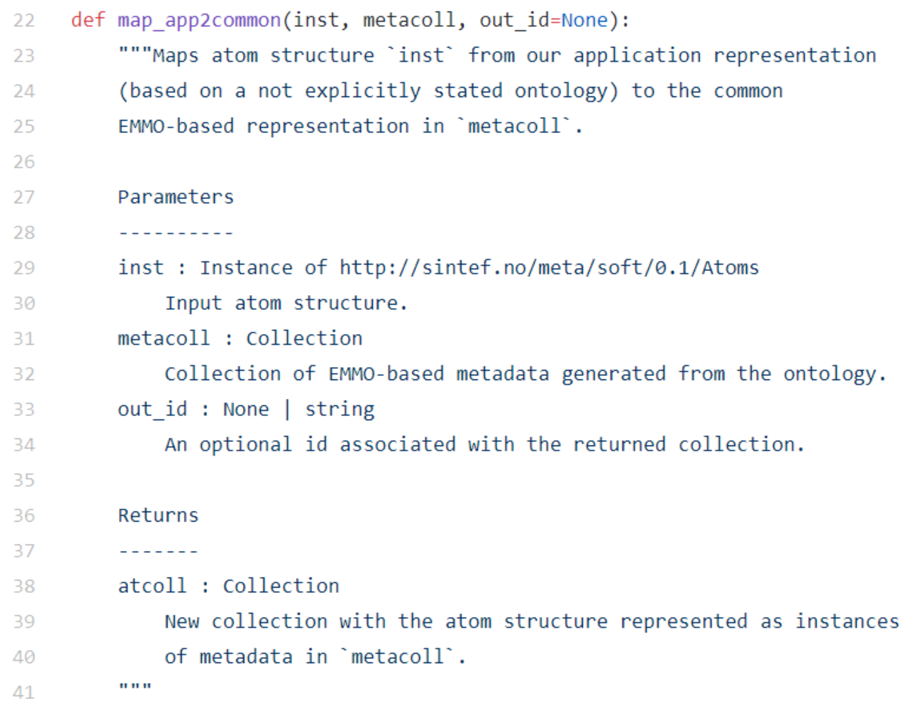


Figure 13. How the DLite class and object factory functions are used to create an ASE atom structure with DLite metadata. Step 2 defines the DLiteAtoms class, which is instantiated in step 3. Filled blue arrows represent function input and output, while solid, dashed and dotted arrows represent *subclass_of*, *instance_of* and *described_by* relations.

The mapping is implemented as a Python function that takes the atomic structure and the metadata representation of the user case ontology as input and returns the atomic structure represented as a collection of instances of the user case metadata (see Python definition in Figure 14).

When applying this mapping function, we create a collection of instances of the user case metadata (from step 1) and relations between them. As visualised in Figure 15, this collection represents the atomic Al/Fe_4Al_{13} interface structure we loaded in step 3 in terms of the user case ontology.



```python
def map_app2common(inst, metacoll, out_id=None):
    """Maps atom structure `inst` from our application representation
    (based on a not explicitly stated ontology) to the common
    EMMO-based representation in `metacoll`.

    Parameters
    ----------
    inst : Instance of http://sintef.no/meta/soft/0.1/Atoms
        Input atom structure.
    metacoll : Collection
        Collection of EMMO-based metadata generated from the ontology.
    out_id : None | string
        An optional id associated with the returned collection.

    Returns
    -------
    atcoll : Collection
        New collection with the atom structure represented as instances
        of metadata in `metacoll`.
    """
```

Figure 14. Definition and documentation of the mapping function in step 4.

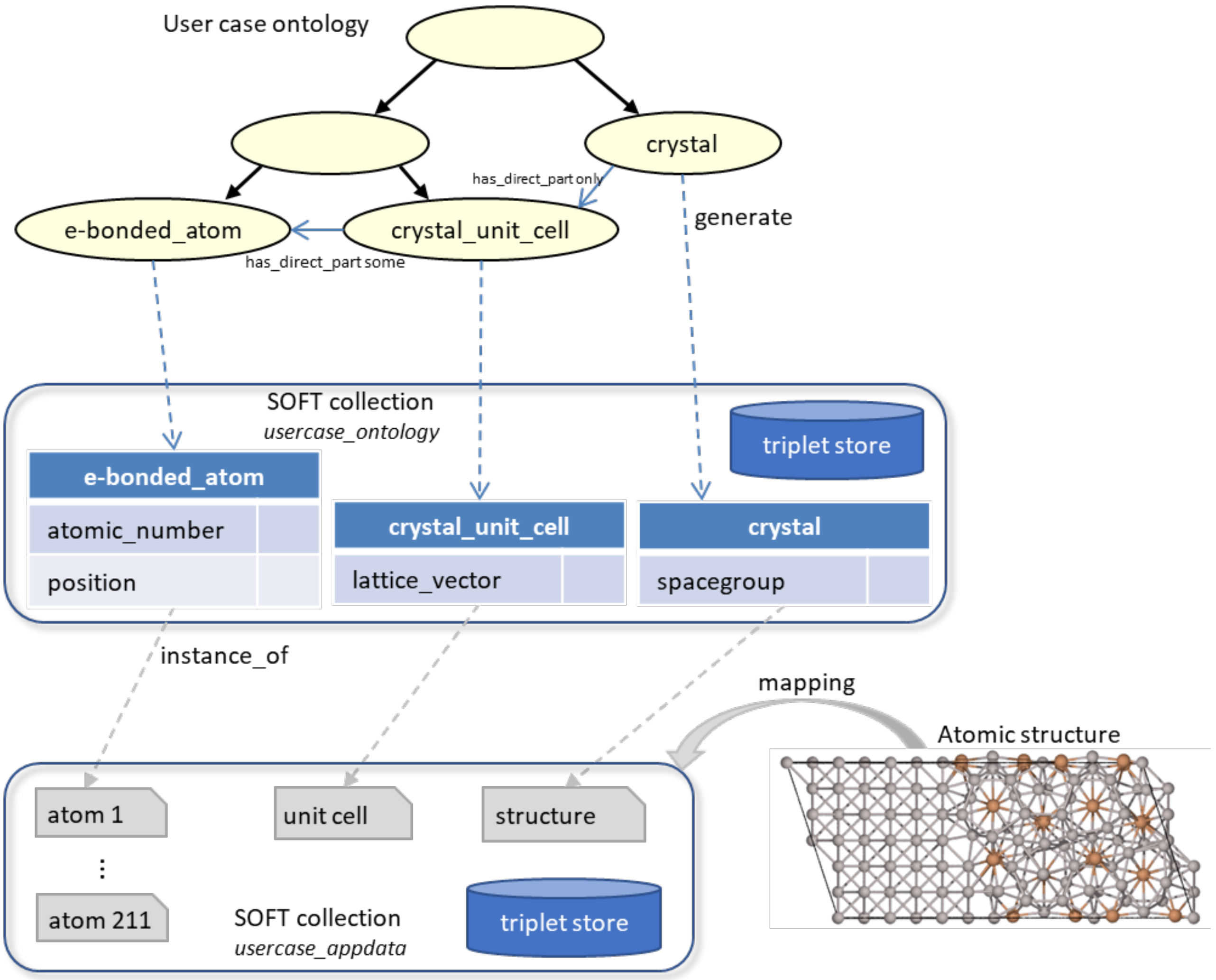


Figure 15. The atomic interface structure mapped into a representation based on the user case ontology. The collection *usercase_appdata* contains instances of metadata generated from the user case ontology. The triplet stores in the collections holds all the relations between the instances in the collection.

3. Conclusions

This documentation presents a hands-on example based on a real user case on how one can use EMMO to achieve interoperability. The demonstration together with the needed tools are made publicly available with permissive licenses on GitHub.

4. References

[1] E. Ghedini, G. Goldbeck, A. Hashibon, G. Schmitz and J. Friis, "European Materials & Modelling Ontology," 2019. [Online]. Available: https://github.com/emmo-repo/EMMO.

[2] J. Friis and F. L. Bleken, "EMMO-python," 2019. [Online]. Available: https://github.com/emmo-repo/EMMO-python.

[3] J. Lamy, "Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies," *Artificial Intelligence,* vol. 80, pp. 11-28, 2017.

[4] T. Bergh, R. Aune, A. Lervik, R. Holmestad and P. E. Vullum, "The Interface between Aluminium and Steel Joined by Cold Metal Transfer - A Transmission Electron Microscopy Study," in *8th Aluminium Surface Science & Technology Symposium*, Helsingør, 2018.

[5] G. Kresse and J. Hafner, *Phys. Rev. B,* vol. 47, p. 558, 1993.

[6] M. Z. Khalid, J. Friis, P. H. Ninive, K. Marthinsen and A. Strandlie, "DFT calculations based insight into bonding character and strength of Fe2Al5 and Fe4Al13 intermetallics at Al-Fe joints," *Procedia Manufacturing,* vol. 15, pp. 1407-1415, 2018.

[7] M. Z. Khalid, J. Friis, P. H. Ninive, K. Marthinsen and A. Strandlie, "Ab-initio study of atomic structure and mechanicals strength of Al/Fe intermetallic interfaces," *In preparation,* 2019.

[8] E. Ghedini, "EMMC-CSA deliverable 2.4 report: Update recommendations for the Review of Materials Modelling, in particular regarding the reference architecture for metadata based semantic interoperability," 2019.

[9] T. Bergh, R. Holmestad and P. E. Vullum, "Scanning Precession Electron Diffraction for Phase Mapping of Intermetallic Compounds," in *Quantitative Electron Microscopy*, Trondheim, 2017.

Authors	Jesper Friis, Francesca Lønstad Bleken, Afaf Saai, Bjørn Tore Løvfall (all SINTEF)

Contributing partners	UNIBO, GCL, FRAUNHOFER, ACCESS

EC-Grant Agreement	723867
Project acronym	EMMC-CSA
Project title	European Materials Modelling Council - Network to capitalize on strong European position in materials modelling and to allow industry to reap the benefits
Instrument	CSA
Programme	HORIZON 2020
Client	European Commission
Start date of project	01 September 2016
Duration	36 months

Consortium		
TU WIEN	Technische Universität Wien	Austria
FRAUNHOFER	Fraunhofer Gesellschaft	Germany
GCL	Goldbeck Consulting Limited	United Kingdom
POLITO	Politecnico di Torino	Italy
UU	Uppsala Universitet	Sweden
DOW	Dow Benelux B.V.	Netherlands
EPFL	Ecole Polytechnique Federale de Lausanne	Switzerland
DPI	Dutch Polymer Institute	Netherlands
SINTEF	Stiftelsen SINTEF	Norway
ACCESS e.V.	ACCESS e.V.	Germany
HZG	Helmholtz-Zentrum Geesthacht Zentrum für Material- und Küstenforschung GMBH	Germany
MDS	Materials Design S.A.R.L	France
QW	QuantumWise A/S	Denmark
GRANTA	Granta Design LTD	United Kingdom
UOY	University of York	United Kingdom
SINTEF	SINTEF AS	Norway
UNIBO	ALMA MATER STUDIORUM – UNIVERSITA DI BOLOGNA	Italy
SYNOPSYS	Synopsys Denmark ApS	Denmark

Coordinator – Administrative information	
Project coordinator name	Nadja ADAMOVIC
Project coordinator organization name	TU WIEN
Address	TU WIEN | E366 ISAS | Gusshausstr. 27-29 | 1040 Vienna | Austria
Phone Numbers	+43 (0)699-1-923-4300
Email	nadja.adamovic@tuwien.ac.at
Project web-sites & other access points	https://emmc.info/

5. Annex

5.1 *Suggestion for a governance plan for EMMO and related domain ontologies and tools*

Below follows a suggestion for a practical plan, and guidelines for a transparent and efficient governance of the development of EMMO, including related domain ontologies and tools. The intension with this section, is to provide input to further discussions regarding the governance structure for EMMO. Further detail will be determined by the EMMO WG of the EMMC.

The basic idea is to use GitHub as the main platform for the governance, which is a leading platform for open source development designed for efficient collaboration and source management.

Suggested guidelines

Organisation or repositories

The GitHub organisation https://github.com/EMMO-repo/ will be considered as the "official" site for development and distribution of EMMO and related domain ontologies and tools. Each of these will have its own git repository within the EMMO-repo organisation. Different people may be involved in the different repositories, but they will have a common governance, to ensure consistent releases and avoid inconsistencies. These repositories may be categorised as:

- EMMO: the core repository for EMMO with the OWL sources for all the top-level modules.
- domain ontologies: ontologies extending EMMO within specific domains, like crystallography, high entropy alloys, CALPHAD or cosmology, that are not of general interest will be managed within separate repositories. These ontologies will follow the conventions outlined for EMMO and never duplicate any class or relation defined in EMMO. Consistency between these ontologies will be strived for via meetings and common overall governance. Currently we have no repositories of this category.
- tools: different tools for working with EMMO and EMMO-based ontologies. EMMO-python is an example of such a tool.

In addition, we foresee that different communities will develop their own EMMO-based third-party domain or application ontologies their own set of tools. Some of these may be brought into the EMMO-repo if it is wished for by both parties and they satisfy the requirements for the "official" domain ontologies and tools.

Governance structure

The upcoming EMMC association will be in charge of EMMO and take responsibility for the governance, maintenance and further development of EMMO. Within the EMMC association, a dedicated *EMMO working group* will coordinate the day-to-day maintenance and development of EMMO. It will also coordinate cross-repository discussions, like resolution of inconsistencies between domain ontologies. Decision about releases, roadmaps, etc will formally be taken by the EMMC governance board, in close dialog with the EMMO working group. The EMMO working group should include the contributors to the different repositories under https://github.com/EMMO-repo/ and be open for any member of EMMC to join.

Releases and versioning

All releases should be versioned according the semantic versioning specifications, see https://semver.org/ and tagged with their version number. The GitHub release feature may be used to associate generated documentation, pre-reasoned versions of the ontology etc. with a release. Only ontologies or tools with version number equal to or larger than 1.0.0 are expected to satisfy all requirements.

Community contributions

Community contributions and new features will be developed in their own branch and incorporated using reviewed pull requests.

Issues and feature requests

GitHub issues will be used for managing issues and feature requests.

Other GitHub features that might be used

- wiki with documentation and roadmap for further development
- packages for distribution of docker images, etc
- statistics

5.2 Definition of user case ontology

The listing below shows the Python script defining the user case ontology. It relies heavily on our Python API for EMMO. When running the script, an ordinary OWL file will be created that can be loaded into Protégé or used with other tools. This OWL file will import EMMO and only include the classes defined below. Hence, there will be no duplication of data.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""An example script that uses EMMO to describe a vertical use case on
welding aluminium to steel and how the thin layer of intermetallic
that are formed at the interface is influencing the overall
properties.  Based on TEM observations using scanning precision
electron diffraction (SPED) the following (simplified) sequence of
intermetallic phases could be established:

    Al | alpha-AlFeSi | Fe4Al13 | Fe2Al5 | Fe

which is consistent with phase stability when the Fe-concentration is
increasing when going from left to right.

In this case study three scales are considered:

  - Macroscopic scale: predicts the overall mechanical behaviour of
    the welded structure during deformation.

  - Microscopic scale: a local crystal plasticity model of a small
    part of the interface.  The constitutive equations were based on
    the results from DFT.  The results from this model was used to
    calibrate decohesion elements for the macroscopic scale.

  - Atomic scale: elastic properties of the individual phases as
    well work of decohesion within and at the interfaces between the
    phases were calculated with DFT [1].  The calculation of work of
    decohesion was performed as a series of rigid steps, providing
    stress-strain relations in both tensile and shear.

References
----------
```

```
[1] Khalid et al. Proc. Manufact. 15 (2018) 1407

"""
from emmo import get_ontology
from owlready2 import sync_reasoner_pellet

# Load EMMO
emmo = get_ontology()
emmo.load()
#emmo.sync_reasoner()

# Create a new ontology with out extensions that imports EMMO
onto = get_ontology('onto.owl')
onto.imported_ontologies.append(emmo)
onto.base_iri = 'http://www.emmc.info/emmc-csa/demo#'

# Add new classes and object/data properties needed by the use case
with onto:

    #
    # Relations
    # =========
    class has_unit(emmo.has_part):
        """Associates a unit to a property."""
        pass

    class is_unit_for(emmo.is_part_of):
        """Associates a property to a unit."""
        inverse_property = has_unit

    class has_type(emmo.has_convention):
        """Associates a type (string, number...) to a property."""
        pass

    class is_type_of(emmo.is_convention_for):
        """Associates a property to a type (string, number...)."""
        inverse_property = has_type

    #
    # Types
    # =====
    class integer(emmo.number):
        pass

    class real(emmo.number):
        pass

    class  string(emmo.number):  #['well-formed']):  #FIXME  Ontology  "emmo-all-
inferred" has no such label: well-formed
        pass

    #
    # Units
    # =====
    class SI_unit(emmo.measurement_unit):
        """Base class for all SI units."""
        pass

    class meter(SI_unit):
        label = ['m']

    class square_meter(SI_unit):
        label = ['mÂò']

    class kilogram(SI_unit):
        label = ['kg']

    class pascal(SI_unit):
        label = ['Pa']


    #
    # Properties
    # ==========
    class position(emmo.physical_quantity):
        """Spatial position of an physical entity."""
        is_a = [has_unit.exactly(1, meter),
                has_type.exactly(3, real)]

    class area(emmo.physical_quantity):
        """Area of a surface."""
        is_a = [has_unit.exactly(1, square_meter),
                has_type.exactly(1, real)]

    class mass(emmo.physical_quantity):
        """Mass of a physical entity."""
        is_a = [has_unit.exactly(1, kilogram),
                has_type.exactly(1, real)]

    class pressure(emmo.physical_quantity):
        """The force applied perpendicular to the surface of an object per
        unit area."""
        is_a = [has_unit.exactly(1, pascal),
                has_type.exactly(1, real)]

    class stiffness_tensor(pressure):
        r"""The stiffness tensor $c_{ijkl}$ is a property of a continuous
        elastic material that relates stresses to strains (Hooks's
        law) according to

            $\sigma_{ij} = c_{ijkl} \epsilon_{kl}$

        Due to symmetry and using the Voight notation, the stiffness
        tensor can be represented as a symmetric 6x6 matrix

            / c_1111  c_1122  c_1133  c_1123  c_1131  c_1112 \
            | c_2211  c_2222  c_2233  c_2223  c_2231  c_2212 |
            | c_3311  c_3322  c_3333  c_3323  c_3331  c_3312 |
            | c_2311  c_2322  c_2333  c_2323  c_2331  c_2312 |
            | c_3111  c_3122  c_3133  c_3123  c_3131  c_3112 |
            \ c_1211  c_1222  c_1233  c_1223  c_1231  c_1212 /

        """
        is_a = [has_unit.exactly(1, pascal),
                has_type.exactly(36, real)]

    class atomic_number(emmo.physical_quantity):
        """Number of protons in the nucleus of an atom."""
        is_a = [has_type.exactly(1, integer)]

    class lattice_vector(emmo.physical_quantity):
        """A vector that participitates defining the unit cell."""
        is_a = [has_unit.exactly(1, meter),
                has_type.exactly(3, real)]
    class spacegroup(emmo.descriptive_property):
        """A spacegroup is the symmetry group off all symmetry operations
        that apply to a crystal structure.

        It is identified by its Hermann-Mauguin symbol or space group
        number (and setting) in the International tables of
        Crystallography. """
        is_a = [has_type.exactly(1, string)]
        pass

    class plasticity(emmo.physical_quantity):
        """Describes Yield stress and material hardening."""
        is_a = [has_unit.exactly(1, pascal),
                has_type.min(2, real)]

    class traction_separation(pressure):
        """The force required to separate two materials a certain distance
        per interface area.  Hence, traction_separation is a curve."""
        is_a = [has_unit.exactly(1, pascal),
                has_type.min(4, real)]

    class load_curve(pressure):
        """A measure for the displacement of a materials as function of the
        applied force."""
        is_a = [has_unit.exactly(1, pascal),
                has_type.min(4, real)]

    #
    # Subdimensional
    # ==============
    class interface(emmo.surface):
        """A 2D surface associated with a boundary.

        Commonly referred to as "interface".
        """
        is_a = [emmo.has_property.exactly(1, area),
                emmo.has_property.exactly(1, traction_separation)]



    #
    # Material classes
    # ================

    # Crystallography-related classes
    # -------------------------------
    class crystal_unit_cell(emmo.mesoscopic):
        """A volume defined by the 3 unit cell vectors.  It contains the atoms
        constituting the unit cell of a crystal."""
        is_a = [emmo.has_spatial_direct_part.some(emmo['e-bonded_atom']),
                emmo.has_property.exactly(3, lattice_vector),
                emmo.has_property.exactly(1, stiffness_tensor)]

    class crystal(emmo.solid):
        """A periodic crystal structure."""
        is_a = [emmo.has_spatial_direct_part.only(crystal_unit_cell),
                emmo.has_property.exactly(1, spacegroup)]

    # Add some properties to our atoms
    emmo['e-bonded_atom'].is_a.append(emmo.has_property.exactly(1,
atomic_number))
```

```
    emmo['e-bonded_atom'].is_a.append(emmo.has_property.exactly(1, mass))
    emmo['e-bonded_atom'].is_a.append(emmo.has_property.exactly(1, position))

    class boundary(emmo.state):
        """A boundary is a 4D region of spacetime shared by two material
        entities."""
        equivalient_to = [emmo.has_spatial_direct_part.exactly(2, emmo.state)]
        is_a = [emmo.has_space_slice.exactly(1, interface),
                emmo.has_property.exactly(1, load_curve)]

    # Continuum
    # ---------
    class phase(emmo.continuum):
        """A phase is a continuum in which properties are homogeneous and can
        have different state of matter."""
        is_a = [emmo.has_property.exactly(1, stiffness_tensor),
                emmo.has_property.exactly(1, plasticity)]

    class rve(emmo.continuum):
        """Representative volume element.  The minimum volume that is
        representative for the system in question."""
        is_a = [emmo.has_spatial_direct_part.only(phase | boundary),
                emmo.has_property.exactly(1, stiffness_tensor),
                emmo.has_property.exactly(1, plasticity)]

    class welded_component(emmo.component):
        """A welded component consisting of two materials welded together
        using a third welding material.  Hence it has spatial direct
        parts 3 materials and two boundaries."""
        is_a = [
            emmo.has_spatial_direct_part.exactly(3, emmo.material),
            emmo.has_spatial_direct_part.exactly(2, boundary),
            emmo.has_property.exactly(1, load_curve)]



# Sync attributes to make sure that all classes get a `label` and to
# include the docstrings in the comments
onto.sync_attributes()


# Sync the reasoner - we use Pellet here becuse HermiT is very slow
sync_reasoner_pellet([onto])


# Save our new EMMO-based ontology.
#
# It seems that owlready2 by default is appending to the existing
# ontology.  To get a clean version, we simply delete the owl file if
# it already exists.
owlfile = 'usercase_ontology.owl'
import os
if os.path.exists(owlfile):
    os.remove(owlfile)
onto.save(owlfile)
```

5.3 Graphical representations of the user case ontology

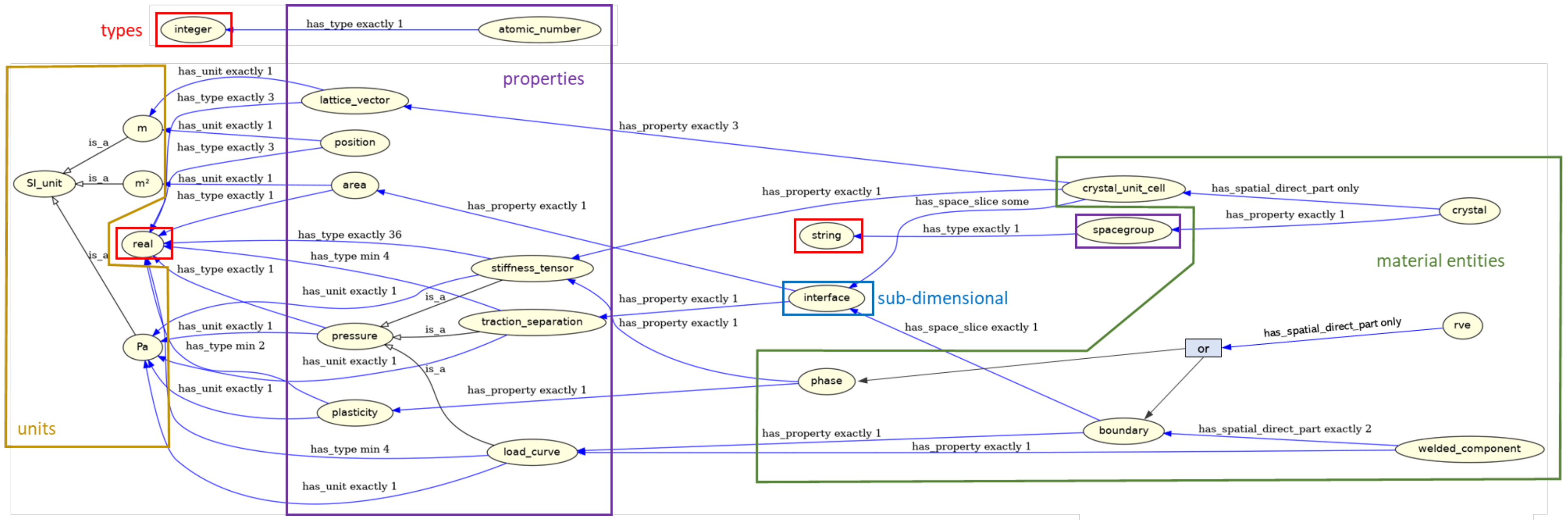


Figure 16. The user case ontology. Only entities that are not in EMMO and relations between them are shown. The colored frames groups together different types of entities. The class construct `rve has_spatial_direct_part only (phase or boundary)` is also shown.

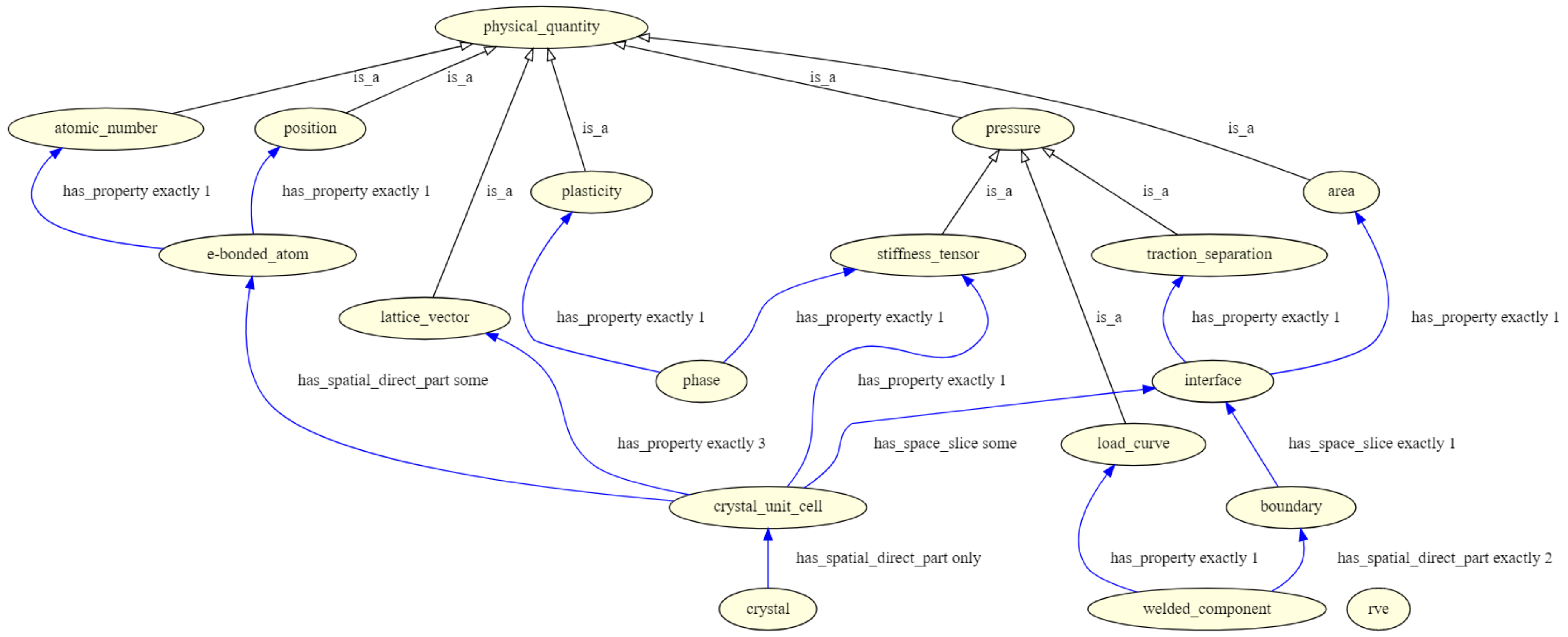


Figure 17. Material entities and related properties. The relation `rve has_spatial_direct_part only (phase or boundary)` is not show.