Working with EMMO extensions, applications etc

Python API

Example 1: EMMO-based user ontology

Example 2: realisation of interoperability

By J. Friis, E. Ghedini, G. Goldbeck, A. Hashibon , G. Schmitz , F.L. Bleken, B.J. Løvfall, A. Saai

EMMO Python API

- Hosted at https://github.com/emmo-repo/
- Open source BSD license

Requires:

- Python 3.5 or higher
- Owlready2 v0.10 (currently issues with v0.13)
- pydot (generation of graphs)
- pandoc (for generation of EMMO documentation)
- java (for reasoning, use pre-reasoned version of EMMO instead)

EMMO Python API

Based on Owlready2

- Python package for ontology-oriented programming
- Selected features
  - transparent access OWL ontologies
  - natural Python representation
    - OWL classes -> Python classes
    - OWL individuals -> Python instances
  - load, modify, save, search (simple + SPARQL), reasoning (via HermiT or Pellet)
  - includes an optimized triplestore/quadstore (via SQLite3)
    - handles large ontologies (>10^9 classes)
- Documentation: https://pythonhosted.org/Owlready2/index.html
- Author: Jean-Baptiste Lamy, LIMICS reseach lab, Sorbonne Paris Cité
- GNU LGPL v3 license

EMMO Python API

EMMO Python package

- A thin EMMO-specific layer on top of Owlready2
  - Makes it easier and more convenient to work with EMMO
  - Generation of graphs
  - Generation of documentation

Working with EMMO via Python

```
thyra:~/prosjekter/MarketPlace$ ipython3
Python 3.6.6 (default, Jul 19 2018, 14:25:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.
```

Importing and loading EMMO (by default pre-reasoned)

```
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#")
```

Accessing class relations

```
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)]
```

Accessing class IRI

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

Search for IRI

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

Search for all properties

```
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]
```

Extending EMMO via Python

Example 1

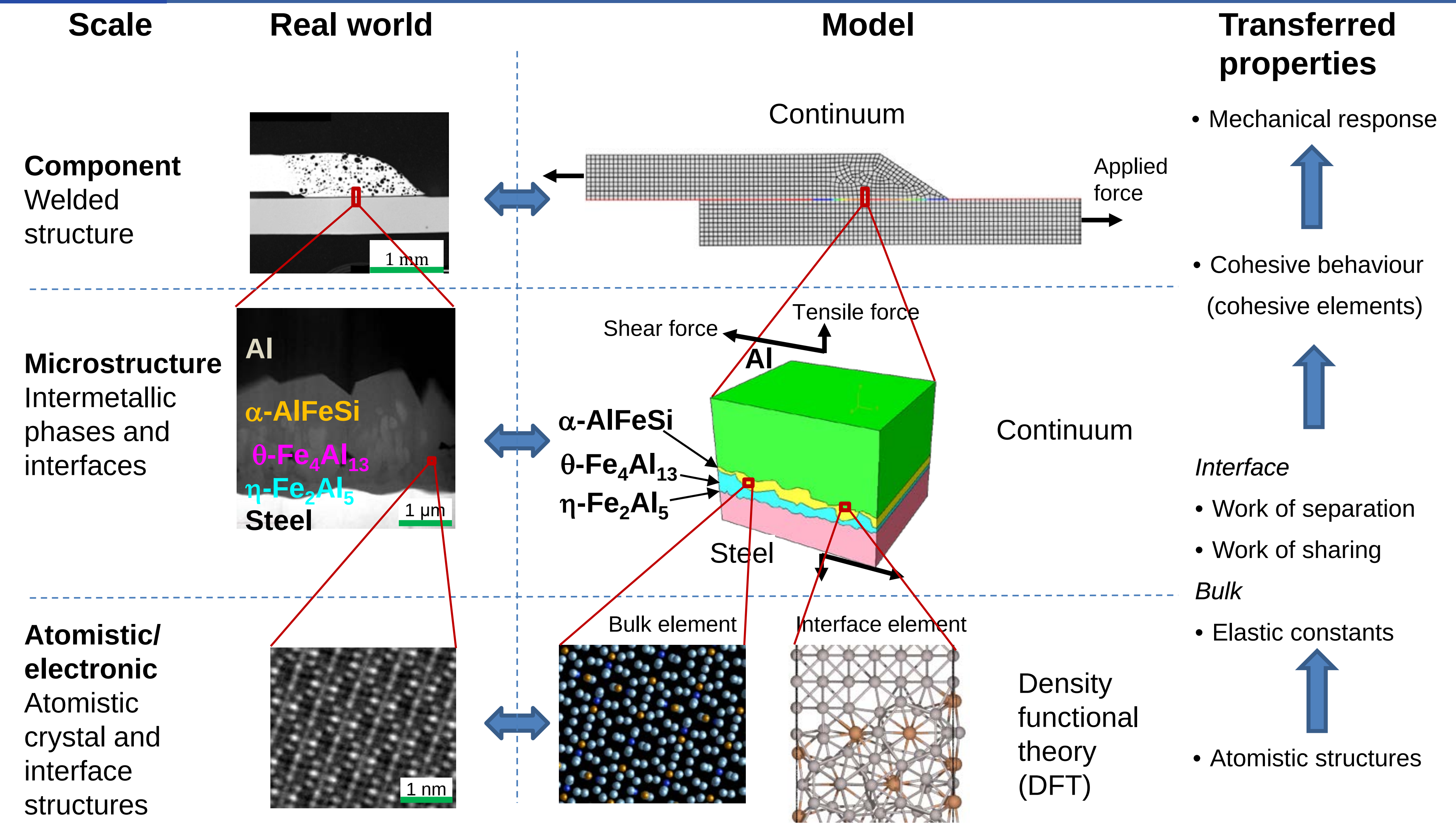
Produces a new owl file: onto.owl

Loading the extended ontology is simple

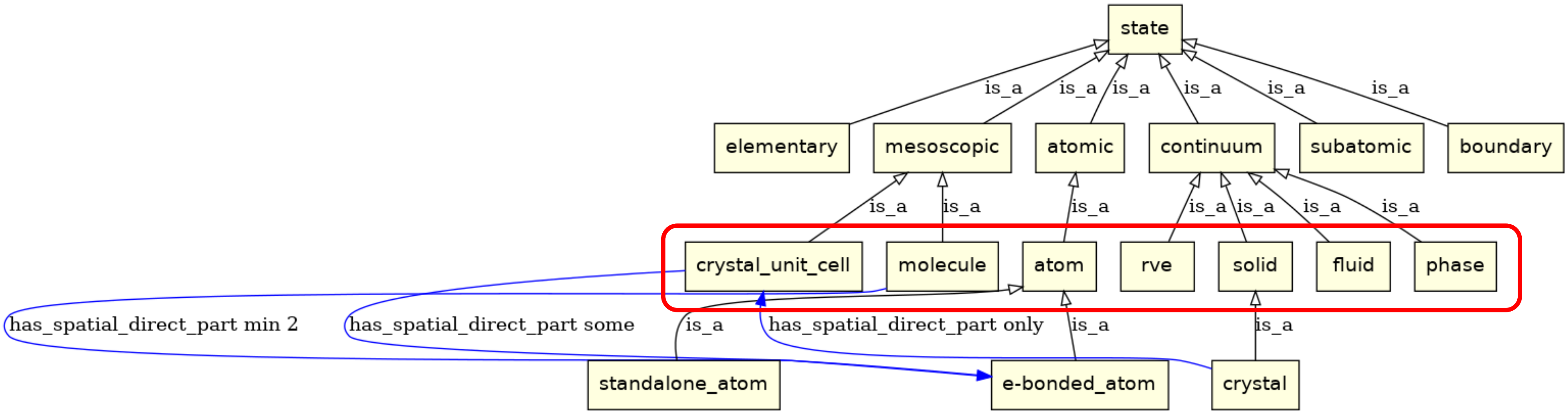
```
In [1]: from emmo import get_ontology

In [2]: onto = get_ontology('onto.owl')

In [3]: onto.load()
Out[3]: get_ontology("onto.owl/onto.owl#")

In [4]: onto.atom.is_a
Out[4]:
[emmo-material.matter,
 emmo-material.atomic,
 emmo-properties.has_property.exactly(1, onto.atomic_number),
 emmo-properties.has_property.exactly(1, onto.atomic_mass),
 emmo-properties.has_property.exactly(1, onto.position),
 emmo-direct.has_spatial_direct_part.exactly(1, emmo-material.electron_cloud),
 emmo-direct.has_spatial_direct_part.exactly(1, emmo-material.nucleus)]

In [5]:
```

```python
#!/usr/bin/env python3
from emmo import get_ontology


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 properties needed by the use case
with onto:

    class crystal(emmo.solid):
        """A periodic crystal structure."""
        label = ['crystal']

    class unit_cell(emmo.descriptive_property):
        """Describes a unit cell in a crystal.  Three cell vectors."""
        label = ['unit_cell']

    class PeriodicAtoms(crystal):
        """Representation of a periodic Atoms class in ASE."""
        equivalent_to = [emmo.has_spatial_direct_part.some(emmo.atom) &
                         emmo.has_property.exactly(1, unit_cell)]

    class atomic_number(emmo.physical_property):
        label = ['atomic_number']


    class atomic_mass(emmo.physical_property):
        label = ['atomic_mass']

    class position(emmo.physical_property):
        label = ['position']

    # Add some properties to our atoms
    emmo.atom.is_a.append(emmo.has_property.exactly(1, atomic_number))
    emmo.atom.is_a.append(emmo.has_property.exactly(1, atomic_mass))
    emmo.atom.is_a.append(emmo.has_property.exactly(1, position))


# Save our new extended version of EMMO
onto.save('onto.owl')
```

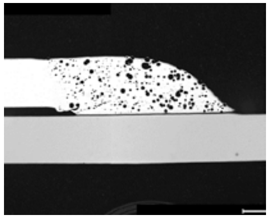
Interoperability user case

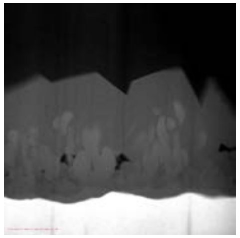


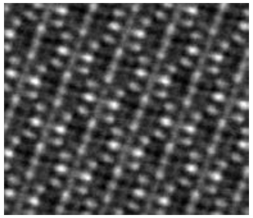
User case ontology
Materials entities



Additional materials classes needed for the user case

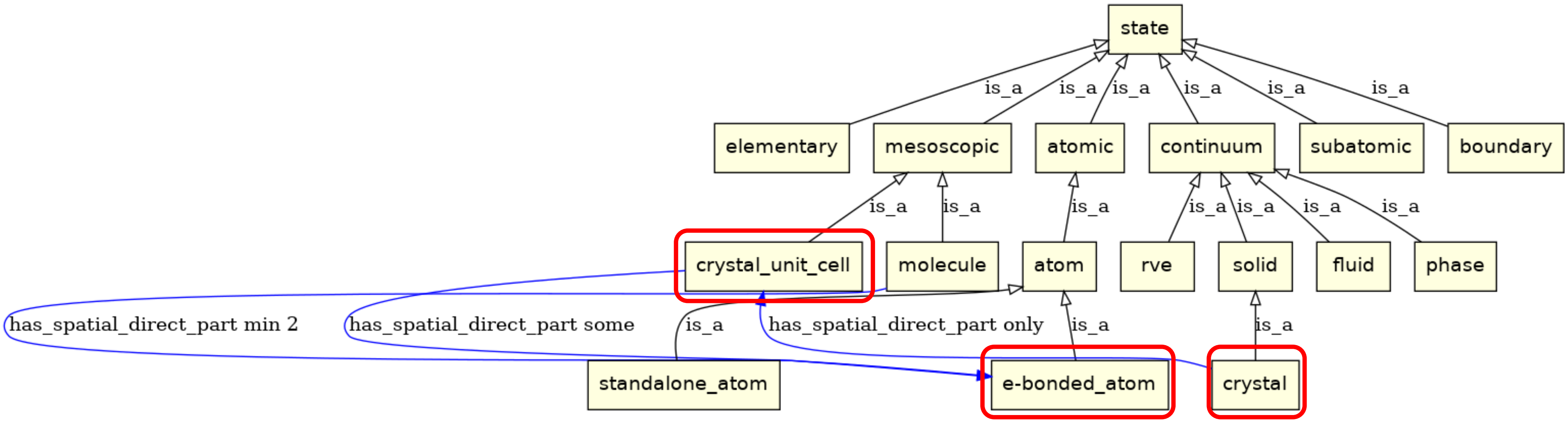


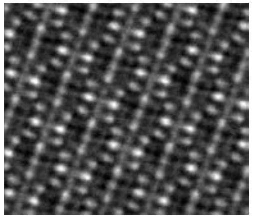




Describing a crystal structure

Materials entities



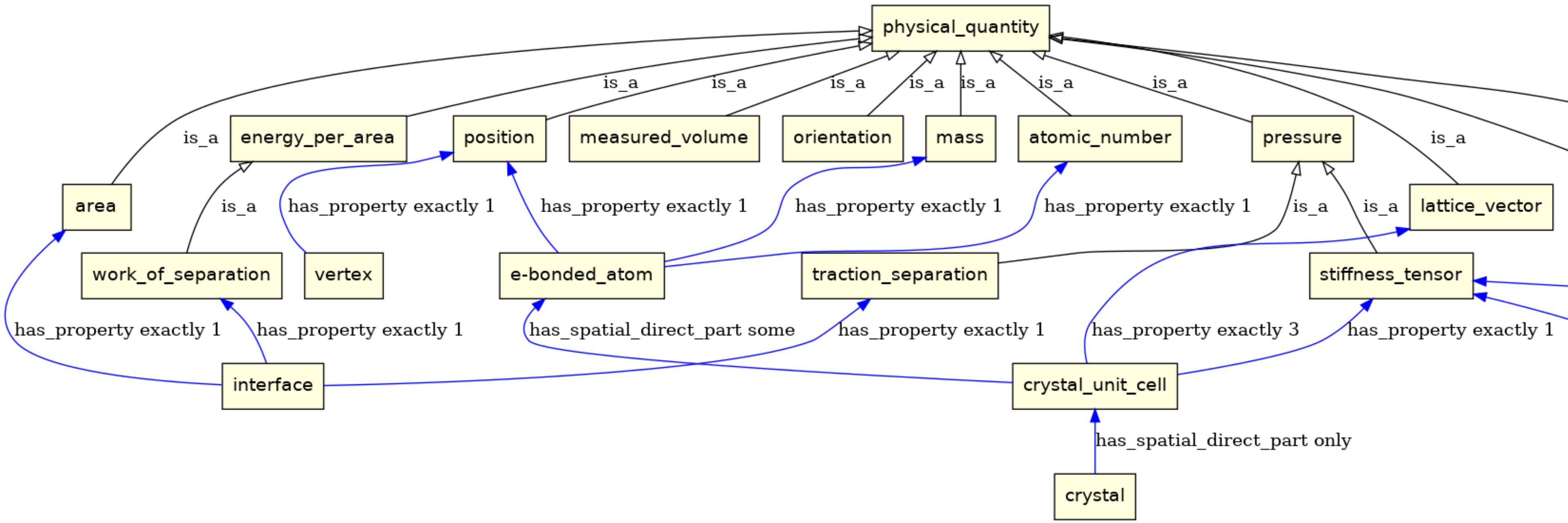


Describing a crystal structure
Material entities

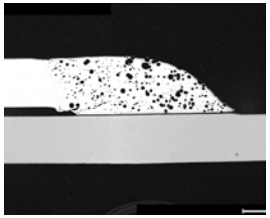
Material entities needed for describing a crystal structure

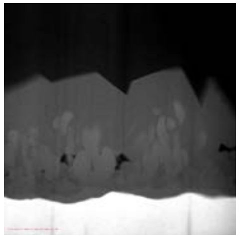
```
# Crystallography-related classes
# -------------------------------
class crystal_unit_cell(emmo.mesoscopic):
    """A volume defined by the 3 unit cell vectors.  It contains the atoms
    constituting the unit cell of a crystal."""
    is_a = [emmo.has_spatial_direct_part.some(emmo['e-bonded_atom']),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

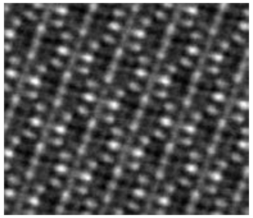
User case ontology
Properties



Properties and related material entities





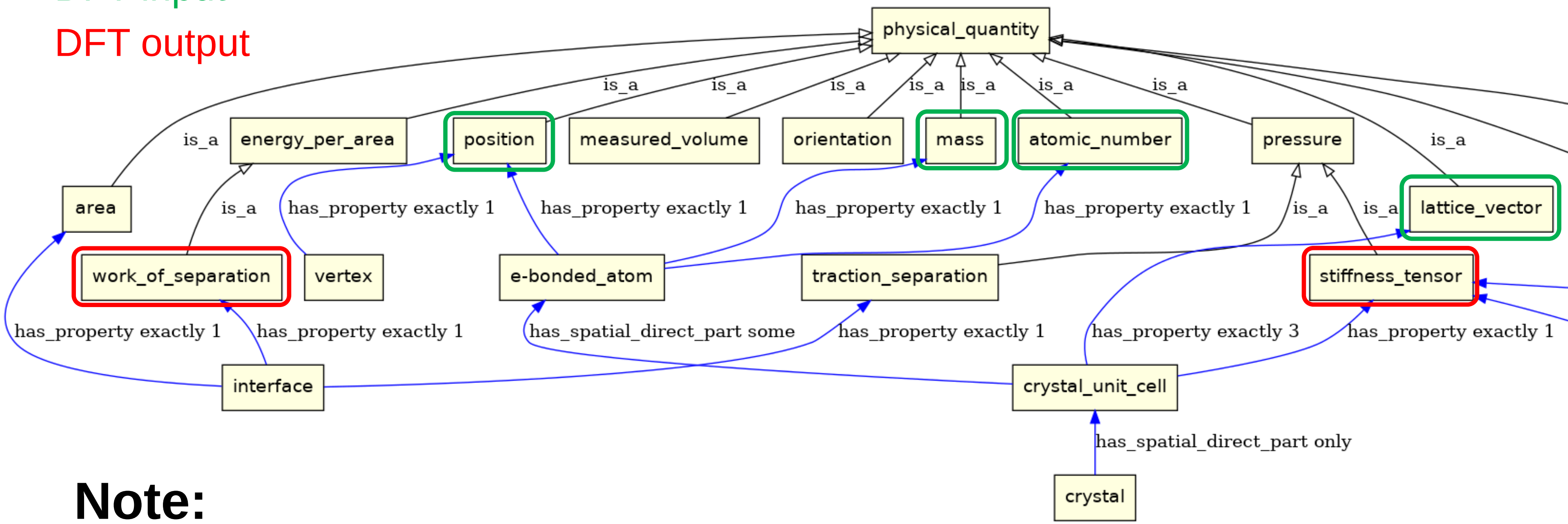


Density functional theory

Properties

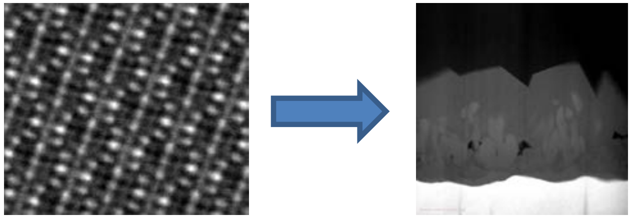
DFT input

DFT output



Note:
`elastic_tensor` is a property of both `crystal_unit` (atomic) and `rve` (continuum)

$\Rightarrow$ vertical interoperability



Describing a crystal structure

Properties

```
class stiffness_tensor(pressure):
    """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.
```

Describing a crystal structure

Properties

```
class stiffness_tensor(pressur
    """The stiffness tensor $c
    elastic material that rela
    law) according to

        $\sigma_{ij} = c_{ijkl

    Due to symmetry and using
    tensor can be represented

        / c_1111  c_1122  c_11
        | c_2211  c_2222  c_22
        | c_3311  c_3322  c_33
        | c_2311  c_2322  c_23
        | c_3111  c_3122  c_31
        \\ c_1211  c_1222  c_1

    """
    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.
```

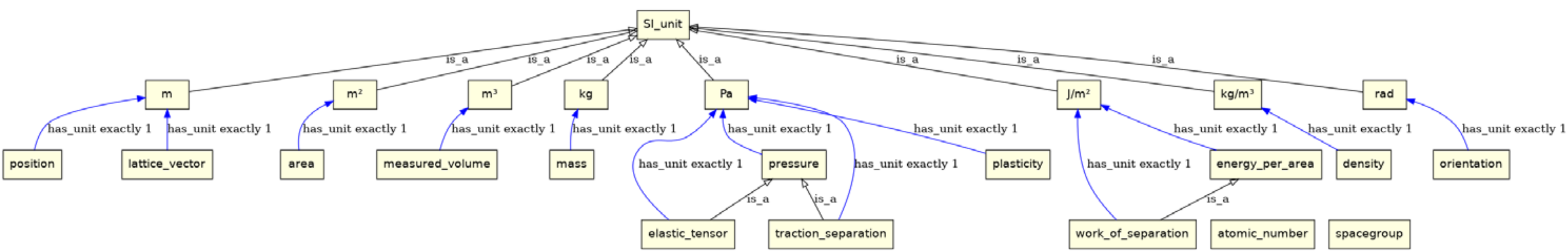
```
#
# Units
# =====
class SI_unit(emmo.measurement_unit):
    """Base class for all SI units."""
    pass

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

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

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

Describing a crystal structure

Properties

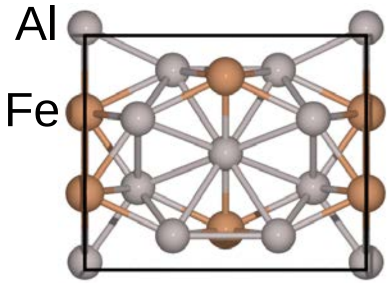
```
class stiffness_tensor(pressure):
    ...{ijkl}$ is a property of a continuous
    ...es stresses to strains (Hooks's
    ... \epsilon_{kl}$
    ...he Voight notation, the stiffness
    ...s a symmetric 6x6 matrix
    ...3  c_1123  c_1131  c_1112 \\
    ...3  c_2223  c_2231  c_2212 |
    ...3  c_3323  c_3331  c_3312 |
    ...3  c_2323  c_2331  c_2312 |
    ...3  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)]
```

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

class real(emmo.number):
    pass

class string(emmo.symbol):
    pass
```

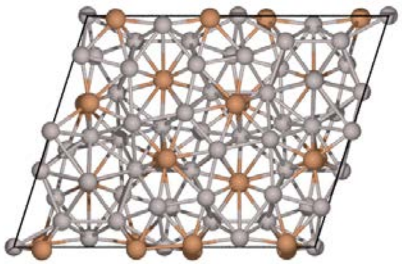



```
class spacegroup(emmo.descriptive_property):
    """A spacegroup is the symmetry group off all symmetry operations
    that apply to a crystal structure.
```

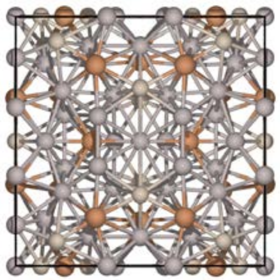
Density functional theory
elastic properties



Fe_2Al_5



Fe_4Al_{13}



alpha

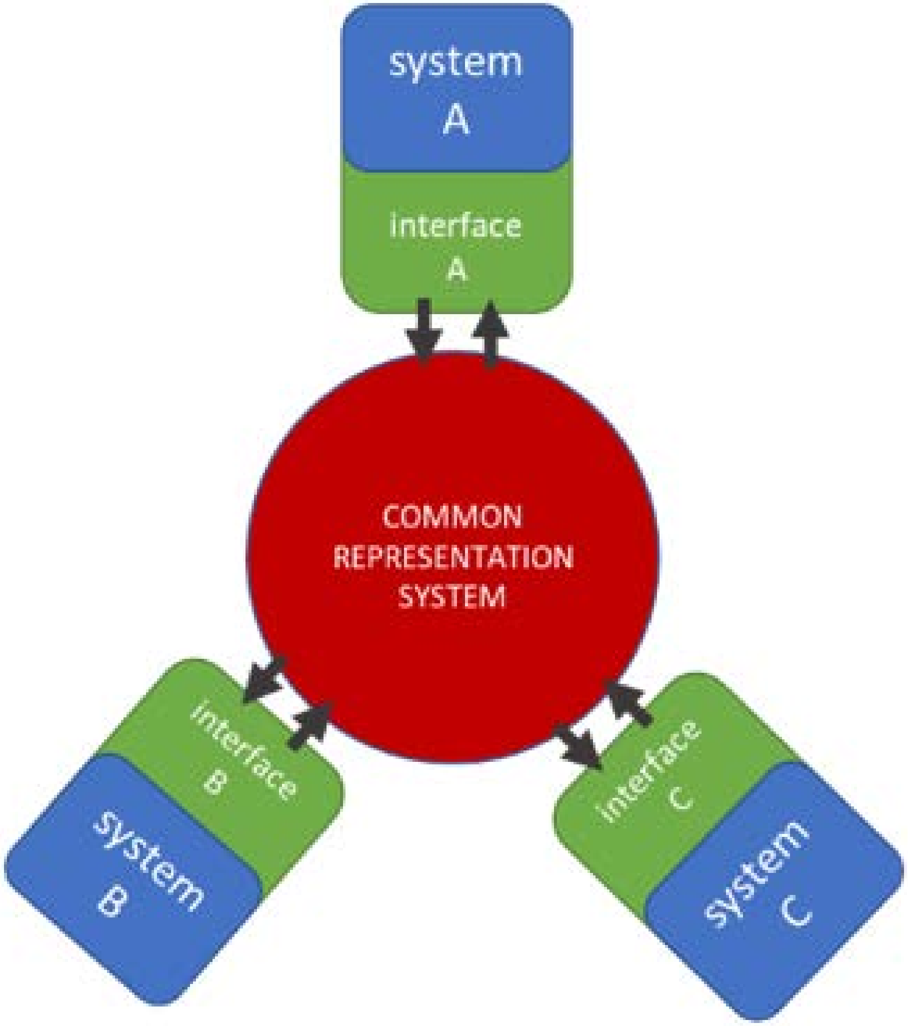
Calculated anisotropic elastic constants

crystal	C_{11}	C_{22}	C_{33}	C_{44}	C_{55}	C_{66}	C_{12}	C_{13}	C_{15}	C_{23}	C_{25}	C_{35}	C_{46}
Fe_2Al_5	213.49	237.49	269.17	88.25	78.25	99.66	77.13	89.43		45.71			
Fe_4Al_{13}	216.3	195.8	219.03	77.09	63.93	76.07	59.70	41.52	-2.75	19.28	-3.61	-3.36	-0.067
Alpha	33.21	38.42	27.32	53.17	53.17	53.17	116.58	116.58		108.03			

Stiffness tensor c_{ijkl} expressed as a 6x6 matrix (from Hooks law $\sigma_{ij} = c_{ijkl}\varepsilon_{kl}$)

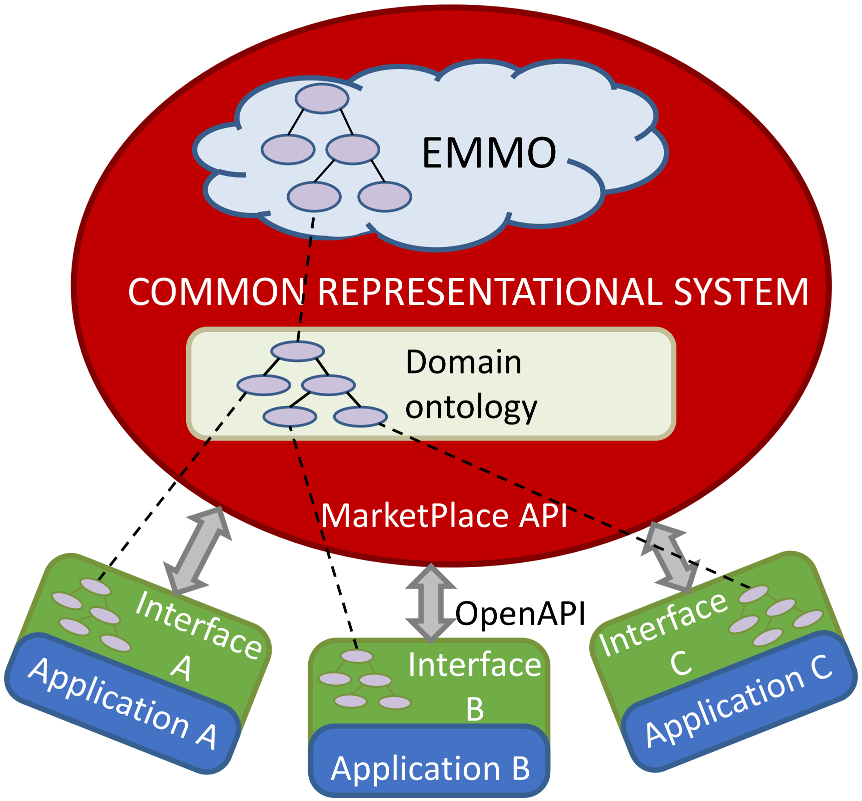
$$c = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\ C_{12} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\ C_{13} & C_{23} & C_{33} & C_{34} & C_{35} & C_{36} \\ C_{14} & C_{24} & C_{34} & C_{44} & C_{45} & C_{46} \\ C_{15} & C_{25} & C_{35} & C_{45} & C_{55} & C_{56} \\ C_{16} & C_{26} & C_{36} & C_{46} & C_{56} & C_{66} \end{bmatrix} = \begin{bmatrix} c_{1111} & c_{1122} & c_{1133} & c_{1123} & c_{1131} & c_{1112} \\ c_{2211} & c_{2222} & c_{2233} & c_{2223} & c_{2231} & c_{2212} \\ c_{3311} & c_{3322} & c_{3333} & c_{3323} & c_{3331} & c_{3312} \\ c_{2311} & c_{2322} & c_{2333} & c_{2323} & c_{2331} & c_{2312} \\ c_{3111} & c_{3122} & c_{3133} & c_{3123} & c_{3131} & c_{3112} \\ c_{1211} & c_{1222} & c_{1233} & c_{1223} & c_{1231} & c_{1212} \end{bmatrix}$$

Realising interoperability
Example 2

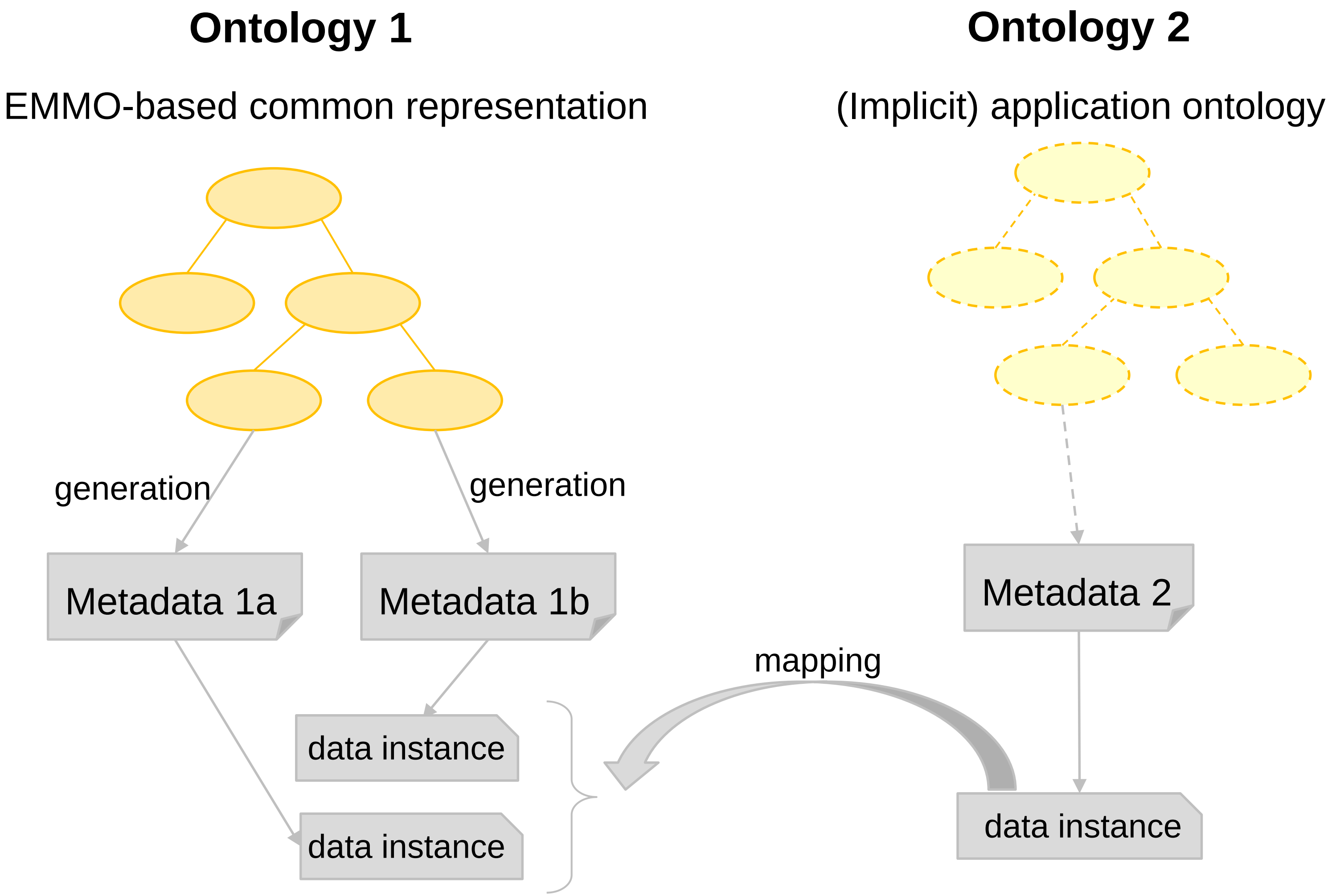


Realising interoperability
Example 2

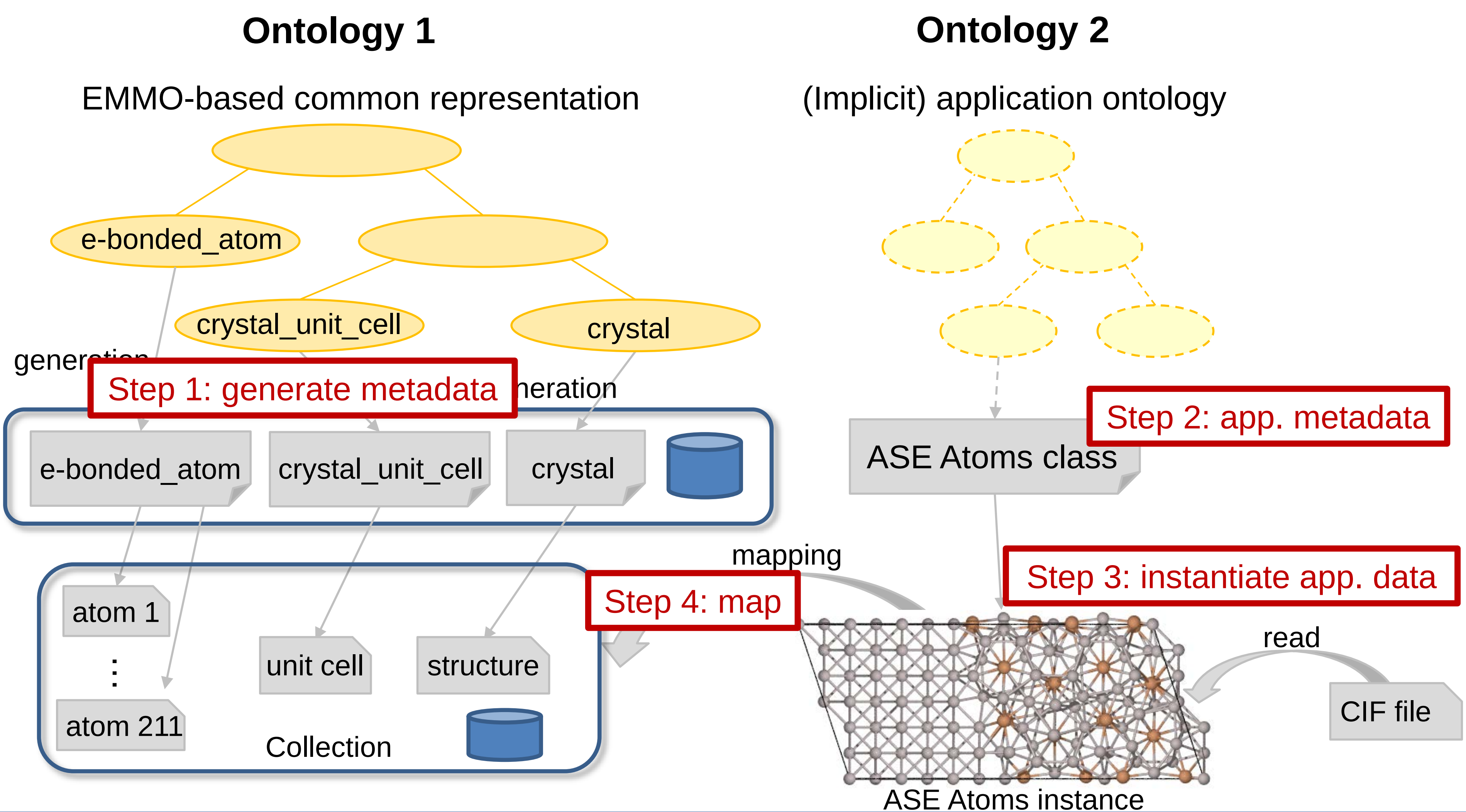
- **Aim:** use common ontology to realise interoperability between applications
- **How**: map between common and application ontology
- **Approach**: use metadata framework (for practicality)
  1. generate metadata from common ontology
  2. define application metadata (from implicit ontology)
  3. instantiate application data
  4. map application data to instance of the common metadata



Realising interoperability
Generic example



Realising interoperability
Generic example



1. Generate metadata

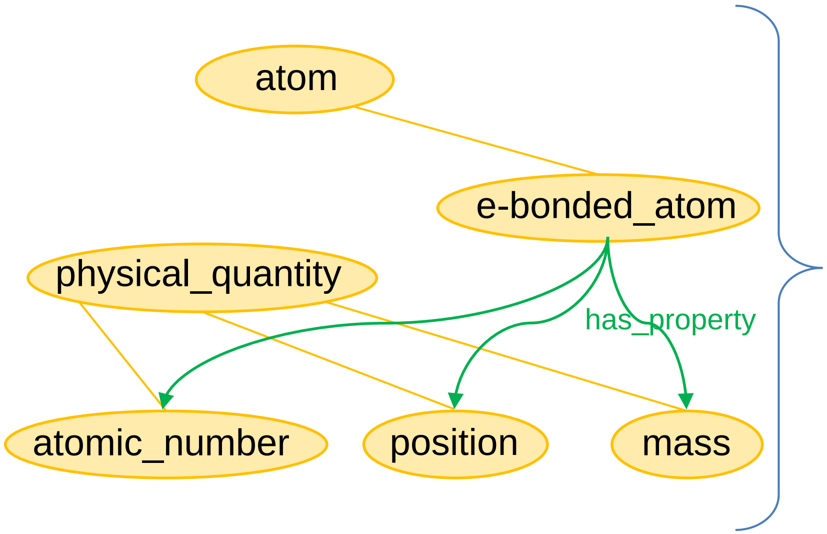
Entity: *"Something that exists by itself, something that is separate from other things"*

Source: Merriam-Webster

About the metadata framework used here

- C-implementation of SOFT (dlite)
- data-driven (property graph)

could equally well have used something else…

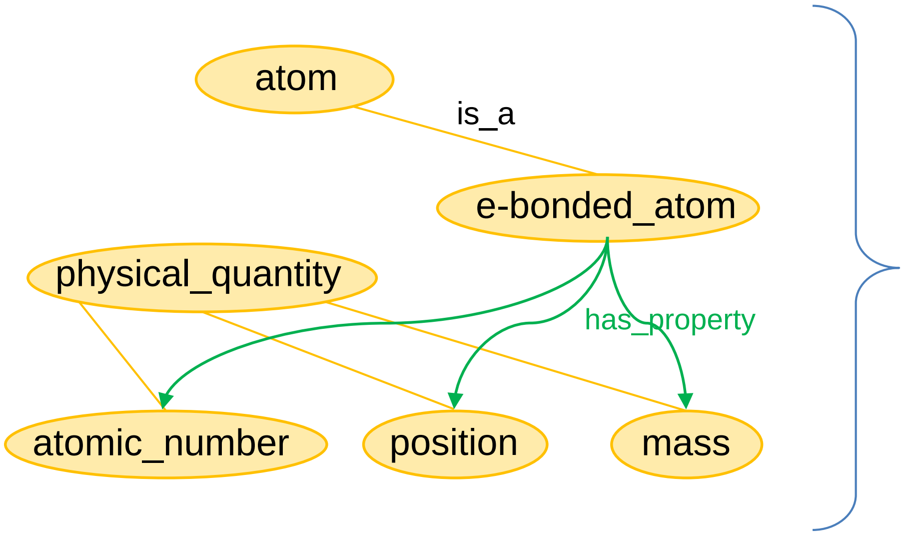


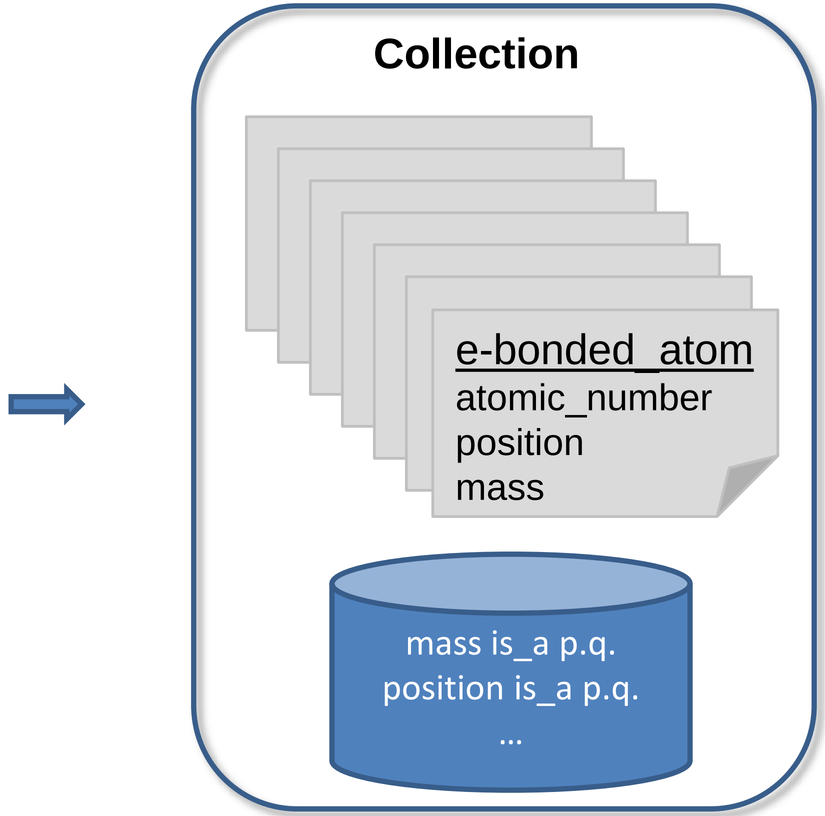
entity example				
Name	e-bonded_atom			
Version	0.1			
Namespace	http://emmc.info/emmc-csa/demo			
Description	An electronic bonded atom that shares at least one electron to the atom_based entity of which is part of.			
Dimensions				
Name	**Description**			
ncoords	Number of coordinates (always 3)			
Properties				
Name	**Type**	**Dims**	**Unit**	**Description**
atomic_number	int	-	-	Number of protons.
mass	float	-	u	Mass of this atom.
position	float	ncoords	Å	Position of this atom.

1. Generate metadata

Mapping of concepts

1. OWL classes → metadata entities
2. EMMO properties → entity properties
3. all other relations → relations
   (+ restriction, class construct instances)





2. Define application metadata

```python
# Create an ASE Atoms subclass that also inherits from dlite atoms.json
DLiteAtoms = dlite.classfactory(ase.Atoms, url='json://atoms.json?mode=r#')
```

atoms.json


```json
{
    "name": "Atoms",
    "version": "0.1",
    "namespace": "http://sintef.no/meta/soft",
    "description": "An ASE Atoms object",
    "dimensions": [
        {
            "name": "natoms",
            "description": "Number of atoms"
        },
        {
            "name": "ncellvecs",
            "description": "Number of cell vectors. Always 3"
        },
        {
            "name": "ncoords",
            "description": "Number coordinates. Always 3"
        },
        {
            "name": "npair",
            "description": "Number in a (key-value) pair.  Always 2"
        },
        {
            "name": "ninfo",
            "description": "Number of info items."
        }
    ],
    "properties": [
        {
            "name": "positions",
            "type": "double",
            "dims": ["natoms", "ncoords"],
            "unit": "Ångström",
            "description": "Atomic positions in Cartesian coordinates."
        },
        {
            "name": "numbers",
            "type": "int64",
            "dims": ["natoms"],
            "description": "Atomic numbers."
        },
```

3. Load atoms

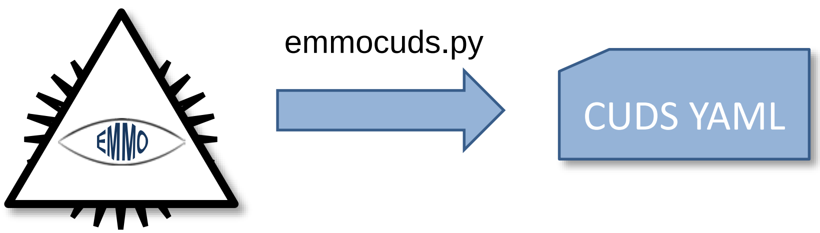
```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Step 3 - load atom structure and represent it using our metadata framework
--------------------------------------------------------------------------
In this step we uses the Atomistic Simulation Environment (ASE) to load
a atomistic Al-Fe4Al13 interface structure from a cif file and
represents it using the metadata defined in step 2.
"""
import ase
import ase.io
from ase.spacegroup import Spacegroup

import dlite

from .step2_define_atoms import DLiteAtoms



# Load atom structure from cif file and convert it to a DLiteAtoms object
at = ase.io.read('../vertical/Al-Fe4Al13.cif')
atoms = dlite.objectfactory(at, cls=DLiteAtoms, instanceid='atoms_Al-Fe4Al13')


# Create a new collection for data instances
coll = dlite.Collection('case_data')
coll.add('Atoms', atoms.dlite_meta)
coll.add('atoms', atoms.dlite_inst)
coll.save('json', 'case_data.json', 'mode=w')
```

4. map Atoms instance to common representation

```
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`.
    """
```

```
# Load metadata collection from step 1
metacoll = dlite.Collection('json://case_metadata.json?mode=r#case_ontology', True

# Load dlite-representation of atoms structure from step 3
coll = dlite.Collection('json://case_data.json?mode=r#case_data', False)
inst = coll.get('atoms')

# Do the mapping
new = map_app2common(inst, metacoll)
```

Using CUDS…



Mappings (EMMO → CUDS)

- class elucidation → description
- classification (`is_a`) → parent
- parthood (`has_part`) → containment
- slicing (`has_subdimension`) → containment
- representations (`has_sign`) → attribute


```yaml
---
VERSION: '1.0'
CUDS: Common Universal Data Structure
Purpose: A representation of EMMO with CUDS
Resources: []
CUDS_ONTOLOGY:
  CUDS_ENTITY:
    definition: Root of all CUDS classes
    parent:
  EMMO:
    definition: The class representing the collection of all the individuals declared
      in this ontology.
    parent: CUBA.CUDS_ENTITY
  SET:
    definition: The class of individuals that 'has_member' some 'item' (i.e. that
      stand for a collection of 'item' individuals).
    parent: CUBA.EMMO
  ITEM:
    definition: "Superclass for all individuals that are subjected to MT MereoTopology.\nThe
      class that collects all the individuals that are member of a set (it\u2019s
      the most comprehensive set individual)."
    parent: CUBA.EMMO
    CUBA.ENCLOSES:
      CUBA.ITEM:
        shape: (:)
  SPACE:
    definition: Pure space entities.
    parent: CUBA.ITEM
    CUBA.HAS_PART:
      CUBA.SPACE:
        shape: (:)
  LINE:
    definition: '1D space entity

      A 1D (space) + 0D (time) substrate.'
    parent: CUBA.SPACE
    CUBA.HAS_PART:
      CUBA.LINE:
        shape: (:)
  SURFACE:
    definition: '2D space entity

      A 2D (space) + 0D (time) substrate.'
    parent: CUBA.SPACE
    CUBA.HAS_PART:
      CUBA.SURFACE:
        shape: (:)
  POINT:
    definition: '0D space entity

      A 0D (space) + 0D (time) substrate.'
    parent: CUBA.SPACE
    CUBA.HAS_PART:
      CUBA.POINT:
        shape: (:)
```


Generated YAML

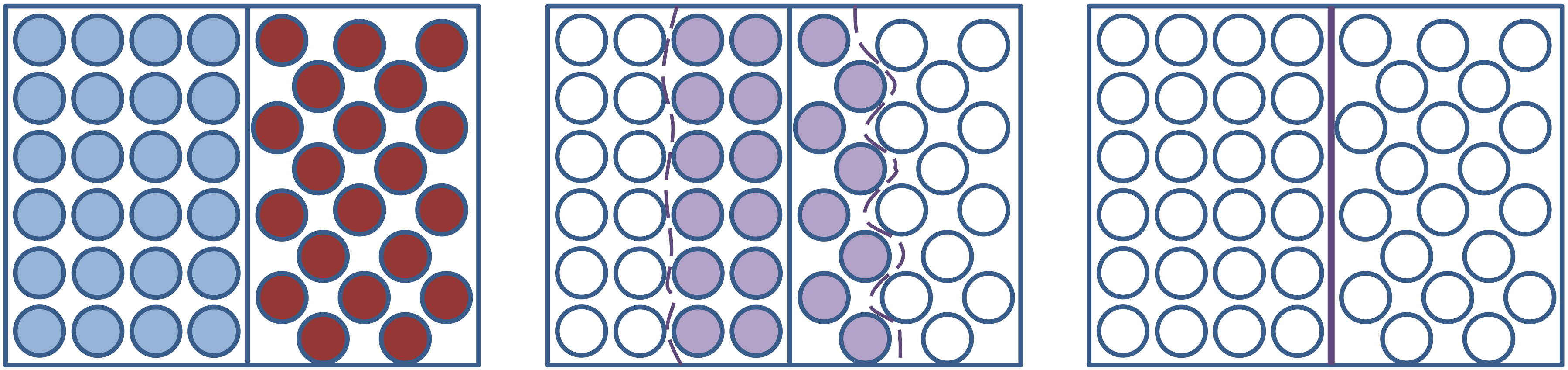
European Materials Modelling Council

EMMC-CSA project has received funding from the European Union's Horizon 2020 research and innovation programme, under Grant Agreement No. 723867.

sfi Centre for Research-based Innovation
The Research Council of Norway
manufacturing

The scientific part of the user case was performed in SFI Manufacturing, a national Norwegian project funded by the Research Council of Norway.

Boundary vs interface

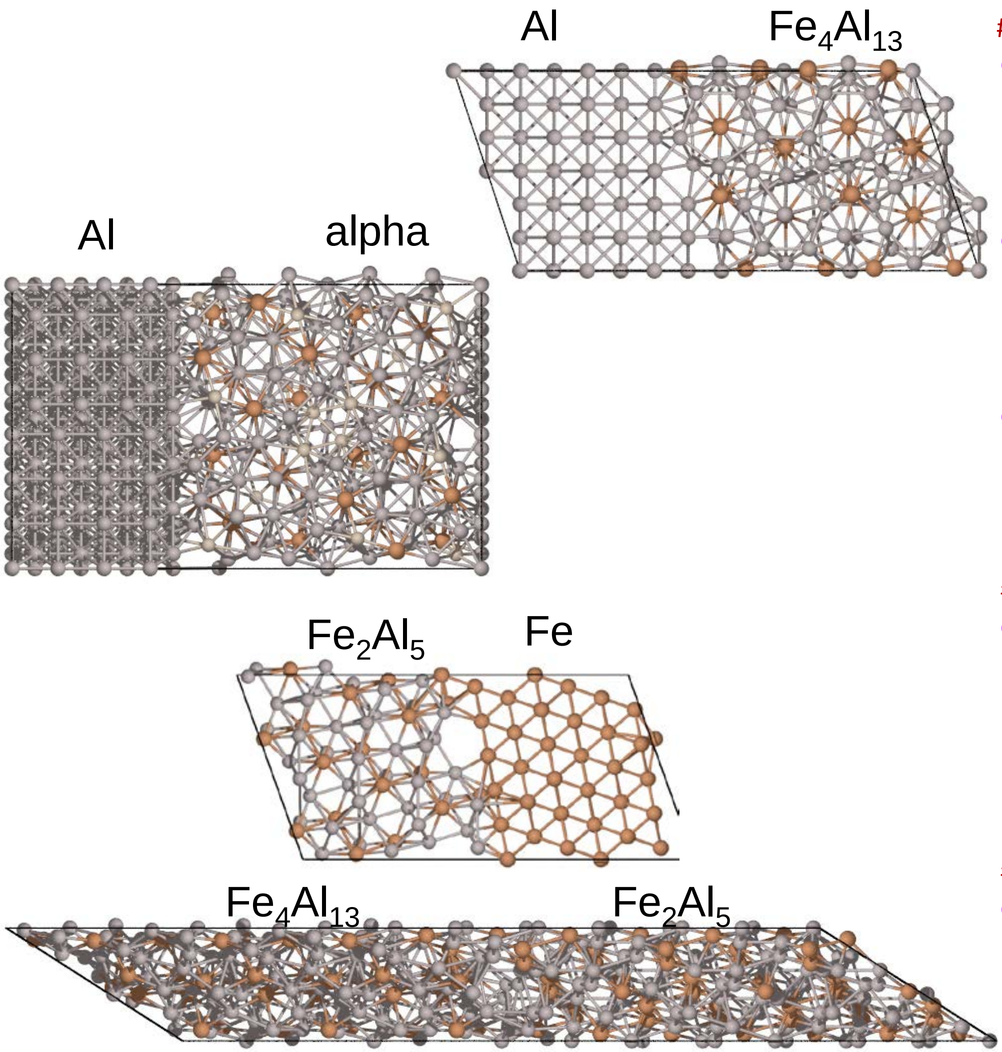


matter **a** (3D + 1D) | matter **b** (3D + 1D)

boundary (matter) (3D + 1D)

interface (world_volume) (2D + 1D)

Density functional theory
Interfacial properties




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

class work_of_separation(energy_per_area):
    """The work required to separate two materials per boundary area."""
    is_a = [has_unit.exactly(1, joule_per_square_meter),
            has_type.exactly(1, real)]

class traction_separation(pressure):
    """The work required to separate two materials per boundary area."""
    is_a = [has_unit.exactly(1, pascal),
            has_type.exactly(1, real)]

# Sub-dimensional classes
class interface(emmo.surface):
    """A 2D surface associated with a boundary."""
    label = ['interface']
    is_a = [emmo.has_property.exactly(1, area),
            emmo.has_property.exactly(1, work_of_separation),
            emmo.has_property.exactly(1, traction_separation)]

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