Non-standard DL Reasoning: Modularisation and Debugging

Oscar Corcho (*)
Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo sn
28660 Boadilla del Monte, Madrid

(*) with inputs from: Bijan Parsia, Uli Sattler, Thomas Schneider, Frank Wolter and Matthew Horridge

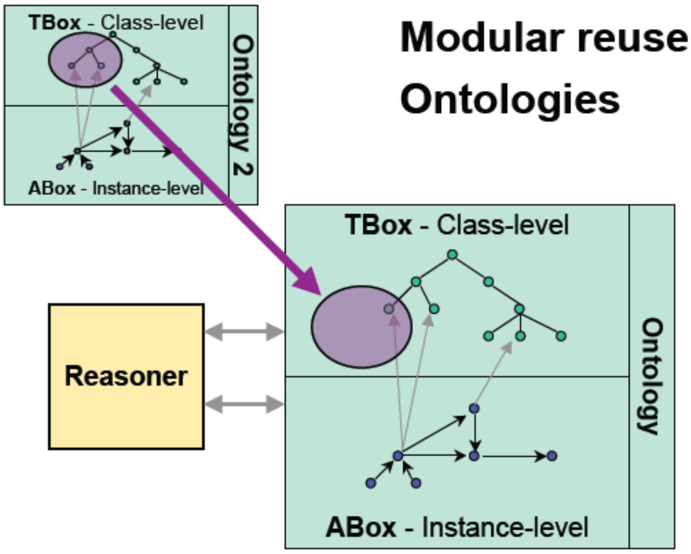
Speaker: Oscar Corcho

Contents

- Modularisation
  - Scenarios for modularisation
  - Modularisation for import/reuse
    - Scenario and a working cycle
    - Properties of the modularisation algorithms
      - Module coverage
      - Safety
  - Modularisation in OWL with Protégé
- Debugging
  - Root and derived unsatisfiable classes
  - Laconic and precise justifications
  - Debugging in OWL with Protégé

Scenarios for Modularisation. Motivation

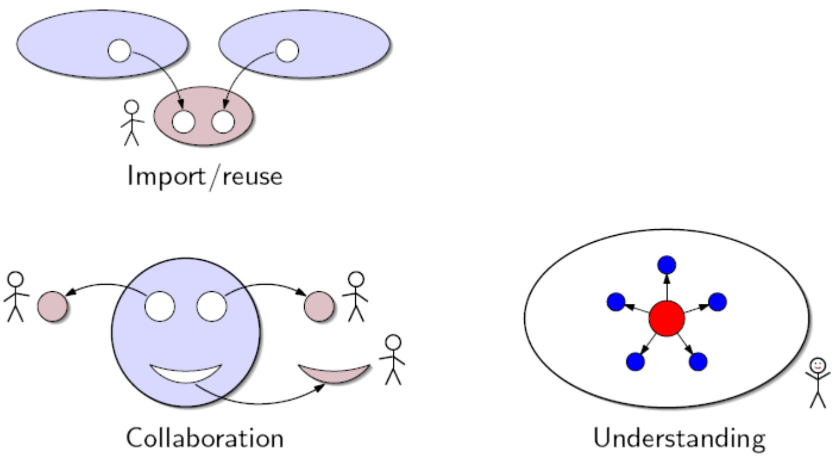
Modular reuse of Ontologies



Reasons for modularisation

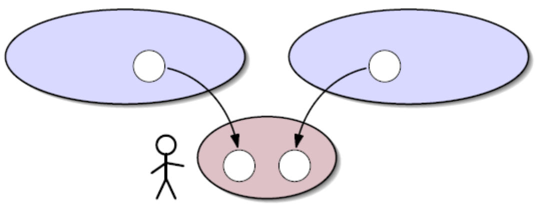
- Common practice in software engineering
  - Modular software development allows for:
    - Importing/reusing modules
    - Collaborative development
    - Understanding the code form the interaction between modules
- Common practice in knowledge engineering
  - Borrow terms from other DL knowledge bases
  - Cover topics that we aren't experts in
  - Enable collaborative development
  - To ensure common understanding
  - To gain insight into its structure & dependencies

Scenarios for modularisation



Scenario 1. Import/Reuse

- “Borrow” knowledge about certain terms from external DL knowledge bases

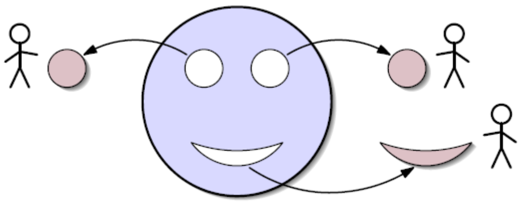


  - Provides access to well-established knowledge
  - Doesn’t require expertise in external disciplines

- This scenario is well-understood and implemented.

Scenario 2. Collaboration

- Collective knowledge base development

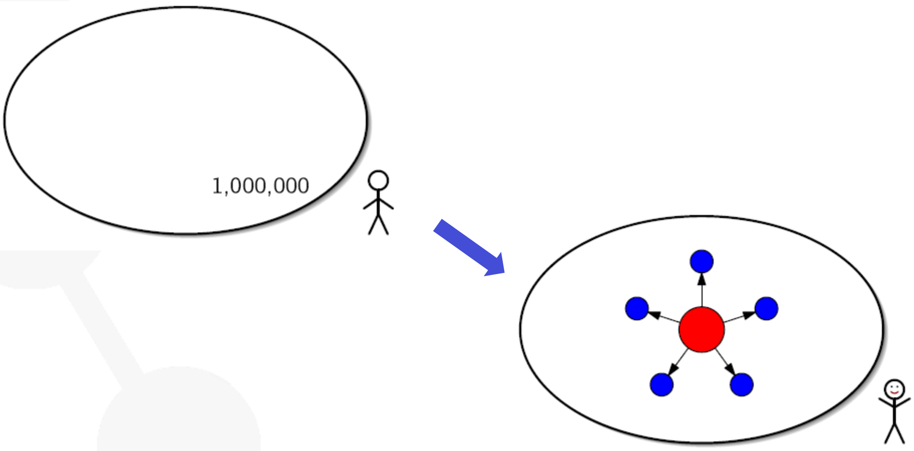


  - Developers work (edit, classify) locally
  - Extra care at re-combination
  - Prescriptive/analytic behaviour

- This approach is understood, but not implemented yet.

Scenario 3. Understanding

- Visualise the modular structure of a DL knowledge base

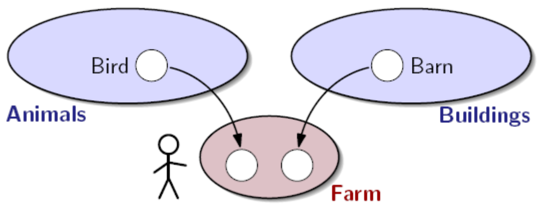


Contents

- Modularisation
  - Scenarios for modularisation
  - Modularisation for import/reuse
    - Scenario and a working cycle
    - Properties of the modularisation algorithms
      - Module coverage
      - Safety
  - Modularisation in OWL with Protégé
- Debugging
  - Root and derived unsatisfiable classes
  - Laconic and precise justifications
  - Debugging in OWL with Protégé

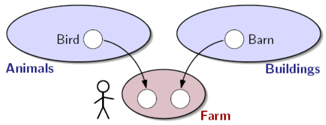
Scenario and main factors to be considered

- Import/reuse a part of external knowledge bases



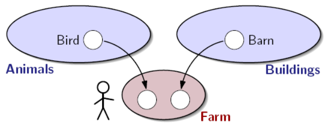
- How much of Animals and Buildings do we need?
  - **Coverage**: Import everything relevant for the chosen terms.
  - **Economy**: Import only what's relevant for them.

Working cycle



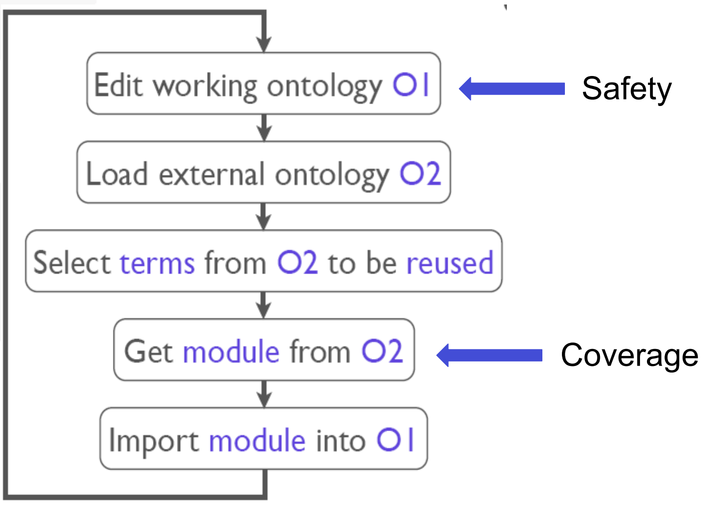
Edit working ontology O1	Farm
Load external ontology O2	Animals
Select terms from O2 to be reused	Animal, feedsOn
Get module from O2	Animals’
Import module into O1	Farm U Animals’

Working cycle



Edit working ontology O1	Farm U Animals’
Load external ontology O2	Buildings
Select terms from O2 to be reused	DuckHousing, Silo
Get module from O2	Buildings’
Import module into O1	Farm U Animals’ U Buildings’

Coverage and Safety



Coverage

- **Goal**: Import everything the external knowledge base knows about the topic that consists of the specified terms.
- **Question**: Which DL axioms do we need to import?

Topic: Fox, Bird, feedsOn

On-topic:

$$\begin{aligned} \text{Fox} &\sqsubseteq \forall\, \text{feedsOn.Bird} \\ \text{Fox} \sqcup \text{Bird} &\sqsubseteq \exists\, \text{feedsOn}.\top \\ \text{Bird} &\sqsubseteq \neg\text{Fox} \\ \text{Bird} &\sqsubseteq \text{Bird} \sqcup \text{Fox} \end{aligned}$$

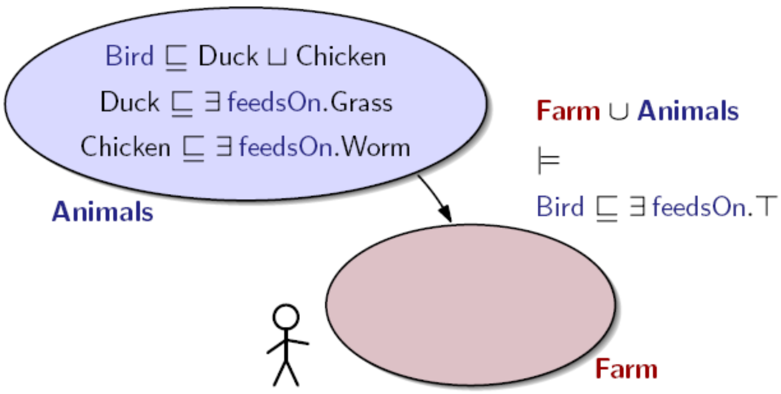
Off-topic:

$$\text{Duck} \sqsubseteq \text{Bird}$$

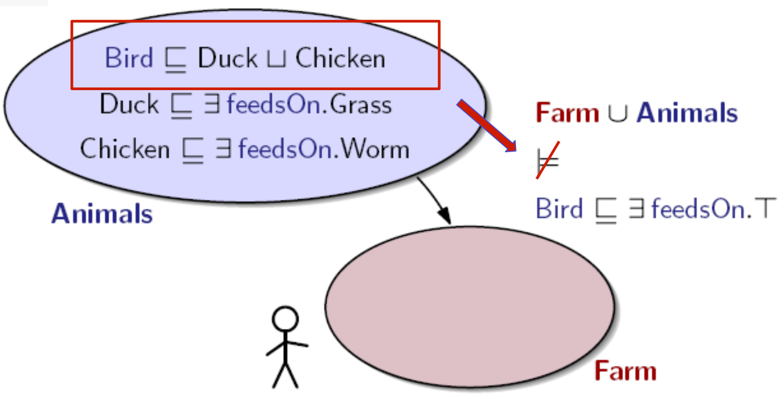
Goal = preserve all on-topic knowledge

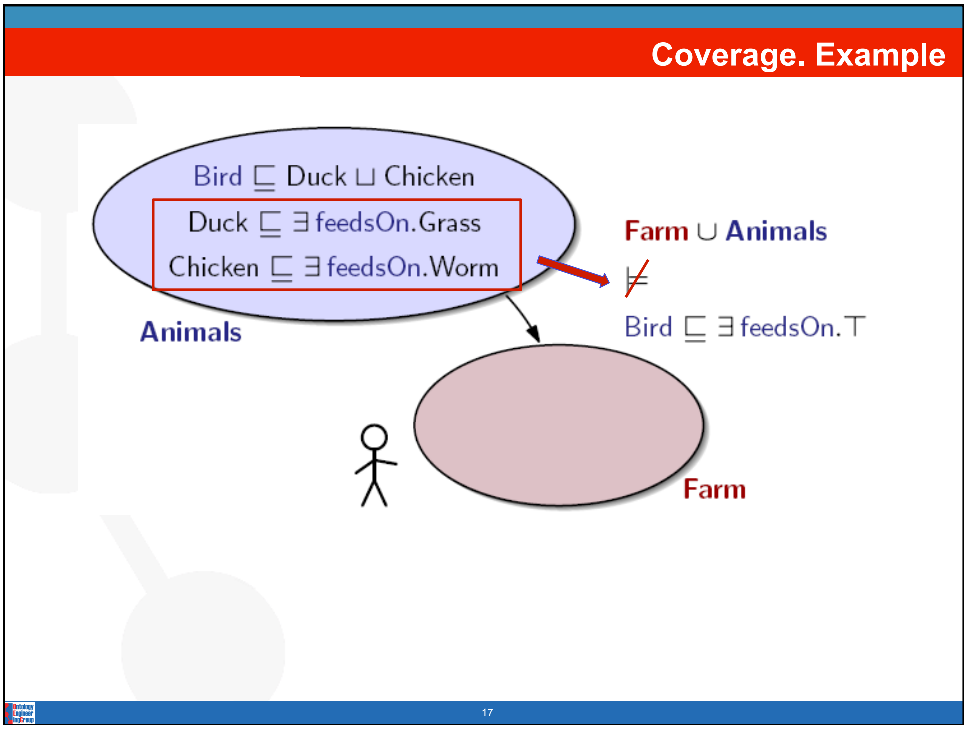
Coverage. Example

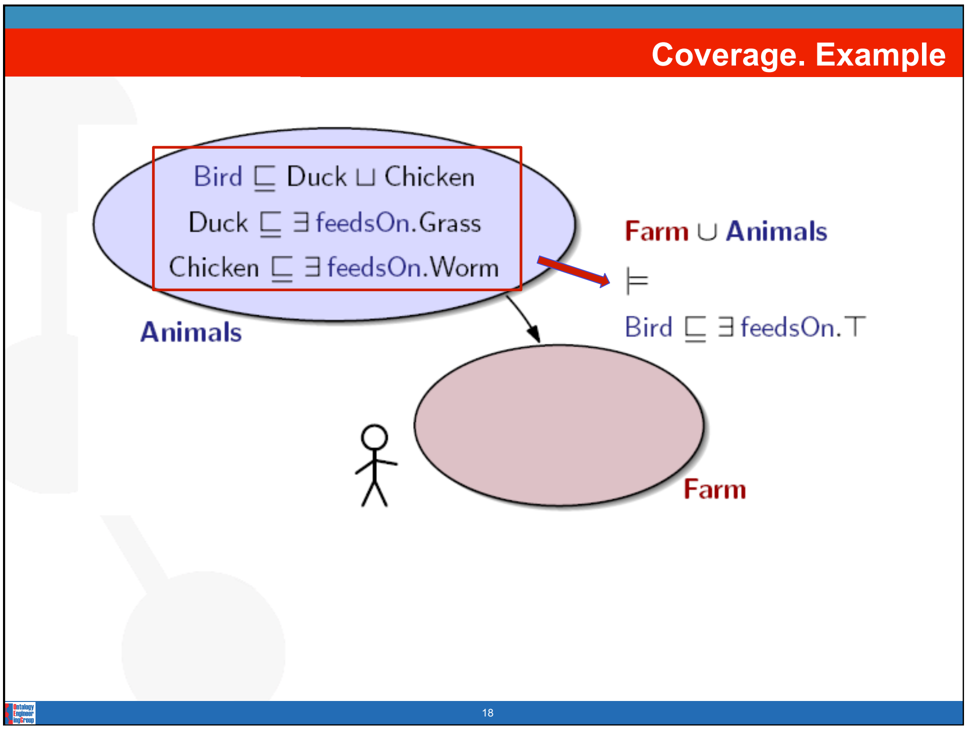
- Let's imagine that we want to import the Animals KB, which contains three axioms, and we want to derive that a bird has to eat something.



Coverage. Example




Coverage. Example
Bird ⊑ Duck ⊔ Chicken
Duck ⊑ ∃ feedsOn.Grass
Chicken ⊑ ∃ feedsOn.Worm
Animals
Farm ∪ Animals
⊭
Bird ⊑ ∃ feedsOn.⊤
Farm
17

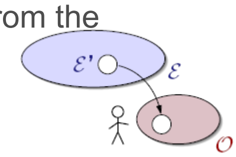

Coverage. Example
Bird ⊑ Duck ⊔ Chicken
Duck ⊑ ∃ feedsOn.Grass
Chicken ⊑ ∃ feedsOn.Worm
Animals
Farm ∪ Animals
⊨
Bird ⊑ ∃ feedsOn.⊤
Farm
18

Coverage. Formal definition and algorithm

- The module Ɛ' covers the knowledge base Ɛ for the specified topic if for all classes A, B built from the specified terms:

  if $O \cup \mathcal{E} \models A \sqsubseteq B$,

  then $O \cup \mathcal{E}' \models A \sqsubseteq B$.



- Coverage = preserving entailments (that is, no difference between using Ɛ or Ɛ')
  - In general, undecidable
- Use a syntactic approximation
  - Fast!
  - Modules are not minimal in size, but guarantee coverage

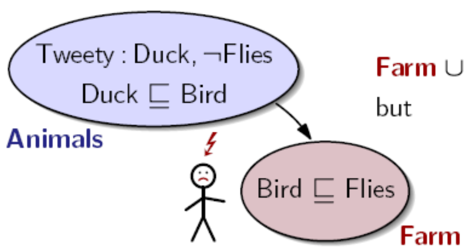
$T \leftarrow$ topic; $M \leftarrow \emptyset$

While there is non-local axiom α w.r.t. $\underline{T \cup \mathrm{sig}(M)}$ do:

$M \leftarrow M \cup \{\alpha\}$ *extended topic*

Safety

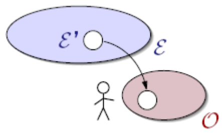
- **Goal**: Don't change the meaning of imported terms.
  - That is, don't add new knowledge about the imported topic.
  - e.g., because you are not an expert in this topic
- **Question**: Which axioms are we allowed to write?



Farm $\cup$ **Animals** $\models$ Bird $\sqsubseteq$ Flies

but **Animals** $\not\models$ Bird $\sqsubseteq$ Flies

- Our knowledge base O uses the imported terms safely if for all classes A, B built from the imported terms:



If	$\mathcal{E}' \not\models A \subseteq B$,
then	$O \cup \mathcal{E}' \not\models A \subseteq B$

- Safety = preserving non-entailments

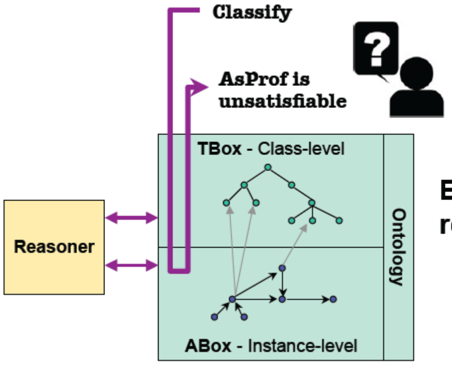
- Safety is also provided by locality (= sufficient condition).

Contents

- Modularisation
  - Scenarios for modularisation
  - Modularisation for import/reuse
    - Scenario and a working cycle
    - Properties of the modularisation algorithms
      - Module coverage
      - Safety
  - Modularisation in OWL with Protégé
- Debugging
  - Root and derived unsatisfiable classes
  - Laconic and precise justifications
  - Debugging in OWL with Protégé

Contents

- Modularisation
  - Scenarios for modularisation
  - Modularisation for import/reuse
    - Scenario and a working cycle
    - Properties of the modularisation algorithms
      - Module coverage
      - Safety
  - Modularisation in OWL with Protégé
- Debugging
  - Root and derived unsatisfiable classes
  - Laconic and precise justifications
  - Debugging in OWL with Protégé



Explanation of reasoning

Root Unsatisfiable Classes

- How do we know which unsatisfiable classes to focus on?
- Example: the TAMBIS ontology contains **144** unsatisfiable classes

- How do we know where to start?
  - The satisfiability of one class may depend on the satisfiability of another class
  - The tools show unsatisfiable class names in red

Root Unsatisfiable Classes

- A class whose satisfiability depends on another class is known as a derived unsatisfiable class
- An unsatisfiable class that is not a derived unsatisfiable class is a root unsatisfiable class
- Root unsatisfiable classes should be examined and fixed first

Justifications

- Justifications are a kind of explanation
  - Justifications are minimal subsets of an ontology that are sufficient for a given entailment to hold
  - Also known as MUPS (Minimal Unsatisfiability-Preserving Sub-TBox), MinAs

$$\mathcal{O} = \{\alpha_1, \alpha_2 \ldots \alpha_n\} \qquad \mathcal{O} \models \eta$$

$$\boxed{\begin{array}{ll} J \subseteq \mathcal{O} & J \models \eta \\ \forall J' \subset J & J' \not\models \eta \end{array}}$$

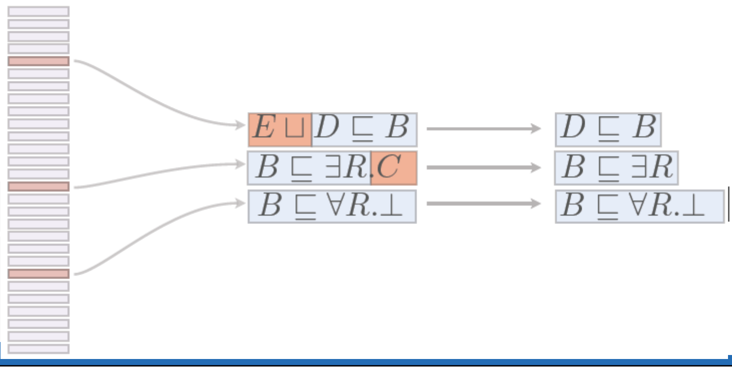
Justifications and root unsatisfiable classes

- There may be multiple justifications for an entailment
- For a given entailment, if there are multiple justifications they may overlap
- Removing one axiom from each justification breaks the justifications so that the entailment is no longer supported by the remaining axioms.

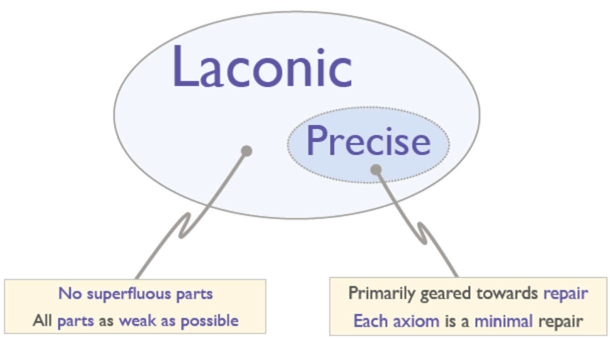
- A class is a **derived unsatisfiable class** if it has a justification that is a superset of a justification for some other unsatisfiable class.
- An unsatisfiable class that is not derived is a **root unsatisfiable class**, i.e., none of its justifications contains a justification of another unsatisfiable class.

Protégé approach to finding justifications

- Black box approach (expand-contract strategy)
  - Create an empty ontology
  - Expand until expression is unsatisfiable
  - Prune until the expression is satisfiable
  - Several optimisations, including the use of modularity
  - Aiming at removing superflous axioms



Regular, Laconic and Precise Justifications



$$\mathcal{O} = \{A \sqsubseteq D \sqcap {=1R.C} \sqcap B, \quad D \sqsubseteq \forall R.C \sqcap F, \quad E \equiv \exists R.C \sqcap \forall R.C\} \models A \sqsubseteq E$$

$$A \sqsubseteq D \sqcap {\geq 1R}$$
$$D \sqsubseteq \forall R.C$$
$$\exists R.C \sqcap \forall R.C \sqsubseteq E$$

Contents

- Modularisation
  - Scenarios for modularisation
  - Modularisation for import/reuse
    - Scenario and a working cycle
    - Properties of the modularisation algorithms
      - Module coverage
      - Safety
  - Modularisation in OWL with Protégé
- Debugging
  - Root and derived unsatisfiable classes
  - Laconic and precise justifications
  - Debugging in OWL with Protégé

33