MDE Assignment 2:
Meta-modeling and conformance checking

Joeri Exelmans
contact: joeri.exelmans@uantwerpen.be

October 16, 2024

1 Introduction

In this assignment, we will manually create (meta-)models and check conformance between them. We will also visualize conformance links using PlantUML.

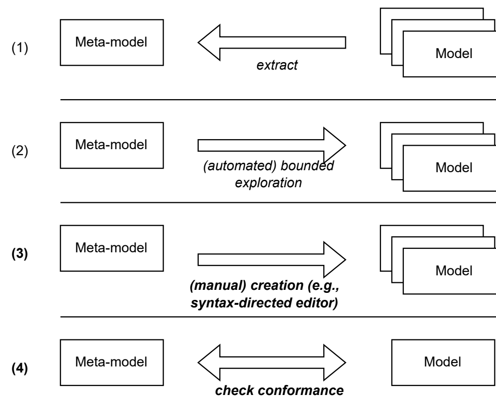


Figure 1: Overview of activities involving (meta-)models.

We will use a custom (meta-)modeling framework, written in Python. In this framework, both models and meta-models are encoded as graphs. Moreover, a meta-model (i.e., a class diagram) can also be seen as a model (i.e., an object diagram), which conforms to a *meta-meta-model* (i.e., a class diagram describing the language of class diagrams). The meta-meta-model conforms to itself. The conformance relations are shown in Figure 2 and Figure 3. Figure 4 shows the full meta-meta-model.

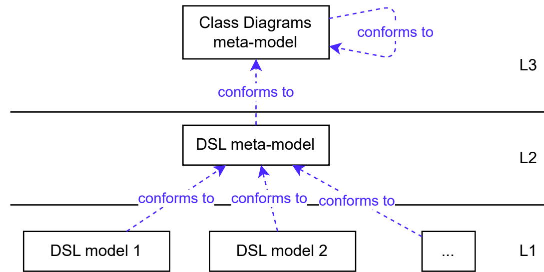


Figure 2: Conformance relations between different levels of meta-ness

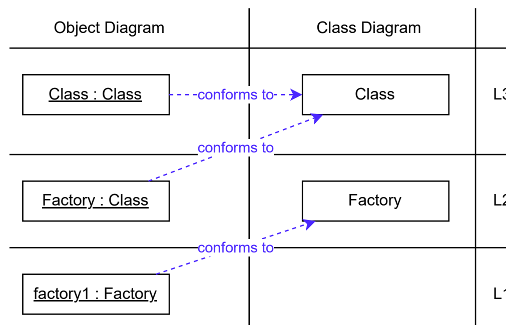


Figure 3: Conformance is always a relation between an object diagram and a class diagram

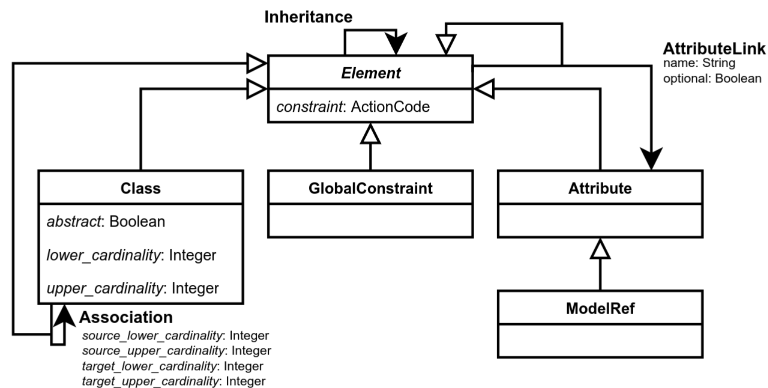


Figure 4: A class diagram conforming to itself (meta-circular).
Conformance-links not shown.
Figure from Andrei Bondarenko's thesis.

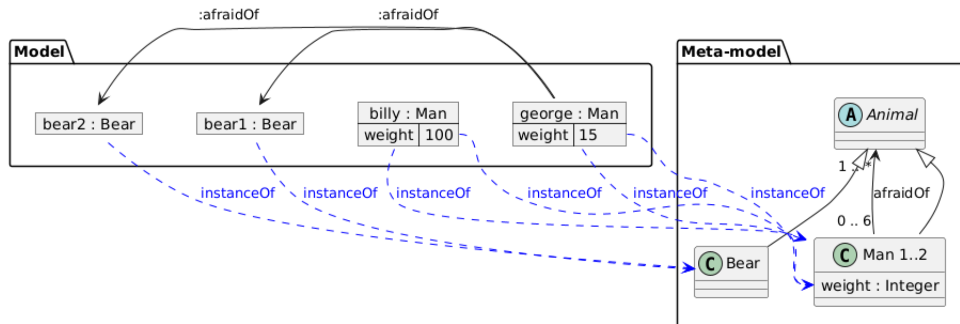


Figure 5: Example: conformance-links between meta-model and model

2 Getting started

- Use git to clone the repo `https://msdl.uantwerpen.be/git/projects/muMLE` or `https://github.com/joeriexelmans/muMLE` (mirror)

  Note: be sure to clone the repo, and not just download the files. By cloning, you can easily merge bug fixes that I publish later on.

- Observe the example in `examples/conformance/woods.py`.

- To run this script, the root of the repo must be in your `PYTHONPATH` environment variable.

  - On Linux/Mac, run the following command:
    `export PYTHONPATH=$PYTHONPATH:/absolute/path/to/repo`

  Also, you'll need the `lark` package from PyPI.

- Run this script. When asked to generate PlantUML, select 'yes' four times, and following the instructions, you should be able to generate a diagram similar to Figure 5.

  This diagram shows the meta-model, model, and conformance-links between their elements. Note that there should also be conformance-links between the links in the model, and the associations in the meta-model, but PlantUML cannot render these.

- Observe the script output:

  - Does the example model conform to the example meta-model or are there errors?
  - Can you explain each constraint violation?
  - Fix the model, so it conforms to the meta-model.
  - Generate PlantUML for the updated model (and save it for your report)

3 Overview of assignment

- Once you have gone through the steps above, copy the file `woods.py` and name it `factory.py`.
- Replace the meta-model by a 'Factory' meta-model. See section 4 for the requirements.
- Create two models:
  - One conforming model
    - you can use a model generated in the previous assignment as inspiration
  - One non-conforming model
    - include a list of conformance-errors in your report

  For each of the models, render PlantUML.
- Submit a ZIP file containing:
  - A small report containing:
    1. your answers to the questions in this document
    2. a brief explanation of how you implemented the various parts of the specification (include code fragments)
  - Your code (`factory.py`)
  - The PlantUML figures of
    - The fixed (conforming) woods-model.
    - Your conforming and non-conforming factory-models.

Practical stuff:

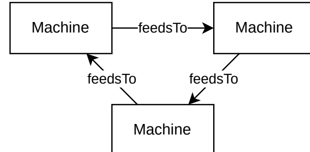
- Students work individually.
- Submission via Blackboard.
- Deadline: Tuesday ~~15~~ 22 October 2024, 23:59.

4 Specification

The specification remains the same as the previous assignment, but a few extra requirements have been added. Gray text indicates requirements from the previous assignment (which still need to be implemented).

- A `Factory` contains at least one `Machine`.
- A `Factory` has at least one `Worker` working for it.

- There are exactly three Shifts: morning, afternoon, and night.
- Workers work one or two Shifts. No Workers work three Shifts (this would be tough for them) and no Workers work zero Shifts.
- A Factory has exactly one Source (to receive parts), and one Sink (to send out assembled products).
- Connections exist between Machines and the Factory's Source/Sink:
  - Every Machine has one or two inputs, that connect to (a) another Machine's output, or (b) the Factory's Source.
  - Every Machine has one or two outputs, that connect to (a) another Machine's input, or (b) the Factory's Sink.
  - Cycles among Machines, such as the following, **are** allowed:

    Cycles between machines are common in industry. For instance, think of a cooling circuit.
  - The Factory's Source and Sink both must be connected to at least one Machine, and never more than two Machines.
  - A Source cannot be connected directly to a Sink.
- Every Machine is operated by zero or more Workers. A Machine that is operated by zero Workers, is considered autonomous (it can function without an operator).
- For every Shift, every non-autonomous Machine (i.e., one that *needs* an operator), must have at least one Worker operating it during that Shift.

- Every Worker has a monthly salary, which must be at least 1000 (could be Euros, Dollars, Belgian Francs, ...) to comply with minimum wage legislation.
- The sum of all montly salaries of all Workers of the same Factory must not exceed 5000. The shareholders are asking for this.

5 Constraint API

When writing constraints, you have the following API at your disposal:

	Local constraint	Global constraint
`this: obj`	current object or link	N/A
`get_name(:obj) : str`	Get name of object or link	
`get_type_name(:obj) : str`	Get type name of object or link	
`get_value(:obj) : int|str|bool`	Get value (only works on Integer, String, Boolean objects)	
`get_target(:link) : obj`	Get target of link	
`get_source(:link) : obj`	Get source of link	
`get_slot(:obj, attr_name:str) : link`	Get slot-link (link connecting object to a value)	
`get_all_instances(type_name:str) : list<(str, obj)>`	Get all instances (tuples (name, object)) of given type	
`get_outgoing(:obj, assoc_name:str) : list<link>`	Get outgoing links of given type	
`get_incoming(:obj, assoc_name:str) : list<link>`	Get incoming links of given type	
`print(**args)`	Python's print function (useful for debugging)	

(Note that `link` is a subtype of `obj`.)

Here are some examples of API usage:

```
# Get the name of the current object or link:
get_name(this)

# Get all instances (objects or links) that are of the same type as
    the current object:
get_all_instances(get_type_name(this))

# Get the value of the 'pay'-slot of the current object:
get_value(get_slot(this, "pay"))


# Print all the unique types that the current object has a '
    hasNeighbor'-link to:

# imperative-style:
types = set()
for neighbor_link in get_outgoing(this, "hasNeighbor"):
  neighbor = get_target(neighbor_link)
  neighbor_t = get_type_name(neighbor)
  ls.add(neighbor_t)
print(types)

# or, more compact but arguably less readable, using a Python list
    comprehension:
print(set(get_type_name(get_target(neighbor_link))
        for neighbor_link in get_outgoing(this, "hasNeighbor")))

# We can count the number of Person-objects in the current model
len(get_all_instances("Person"))

# We can also count the number of 'hasNeighbor'-links:
len(get_all_instances("hasNeighbor"))
```

6 Tips

- In your (meta-)models, you can only refer to things that have already been declared. For instance, the following will fail to parse (with a **cryptic error**):

```
obj1:Type1
```

```
lnk:Link (obj1 -> obj2) # fail
obj2:Type2
```

To fix this, declare 'obj2' first:

```
obj1:Type1
obj2:Type2
lnk:Link (obj1 -> obj2) # good
```

- Any object or link can be named, or unnamed. Example of inheritance link:

  ```
  :Inheritance(Man -> Animal) # unnamed
  bear_inherits_animal:Inheritance (Bear -> Animal) # named
  ```

  Example of object:

  ```
  :Bear { ... } # unnamed
  billy:Man { ... } # named
  ```

  All objects must be uniquely named within the context of the diagram. Unnamed objects get auto-generated unique names behind the scenes. The only drawback of unnamed things is that you cannot explicitly refer to them. For something like an inheritance link, this is not a problem.

- A local constraint defined on a type (e.g., Class, AttributeLink or Association) will be checked on every instance of that type (**and** the type's subtypes). So if a type has 10 instances, its constraint code will run 10 times, each time with a different `this`-object.

  For instance, given the following meta-model:

  ```
  Animal:Class {
      abstract = True;
      constraint = `get_name(this) != "billy"`; # will fail, billy is Animal
  }
  Bear:Class {
      constraint = `get_name(this) != "billy"`; # OK
  }
  :Inheritance (Bear -> Animal)

  Man:Class
  :Inheritance (Man -> Animal)
  ```

  and the following model:

```
george:Man
billy:Man
bear1:Bear
bear2:Bear
```

a conformance check will execute the 'Animal'-constraint 4 times (george, billy, bear1, bear2).

- Every global constraint is checked only once during a conformance check.
- In the concrete syntax, Python code can be surrounded by a single backtick ` or triple backticks ```. When using a single backtick, the raw string of code is passed to the Python parser, which may give problems with the indentation when the code has multiple lines. When using triple backticks, the entire string is de-indented by the amount of indentation on the first non-empty line. Triple backticks are recommended when writing multi-line constraints.