MDE Assignment 2:
Meta-modeling and conformance checking

Vanessa Flügel
Contact: vanessa.flugel@uantwerpen.be

October 8, 2025

You are discouraged but allowed to make use of any GenAI tool in solving the assignment or writing the report (for example, to correct grammatical mistakes) as long as you adhere to the UA Guidelines for students on responsible use of GenAI.

You are required to cite any use of GenAI and describe what portions of the assignment you used it for. Note that a significant part of demonstrating that the learning goals have been achieved, includes being able to explain the relevant concepts in the assignment, explain the design choices in your implementation, and critically discuss your solution.

1 Introduction

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

For this you will use **muMLE**, a custom (meta-)modeling framework, written in Python, that allows us to textually define models. Both models and their meta-models are encoded as graphs, which are then used to check conformance between the model and meta-model. Note that a meta-model is itself a model that conforms to a meta-meta-model. If the meta-model is a class diagram, its meta-meta-model is the language of class diagrams. This meta-meta-model in turn conforms to itself: it is at the meta-circular level. The conformance relations are shown in Figure 1 and Figure 2 shows the full meta-meta-model.

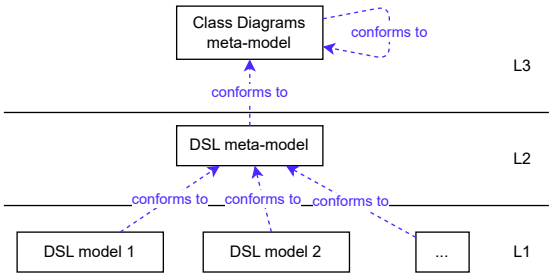

Figure 1: Conformance relations between different levels of meta-ness

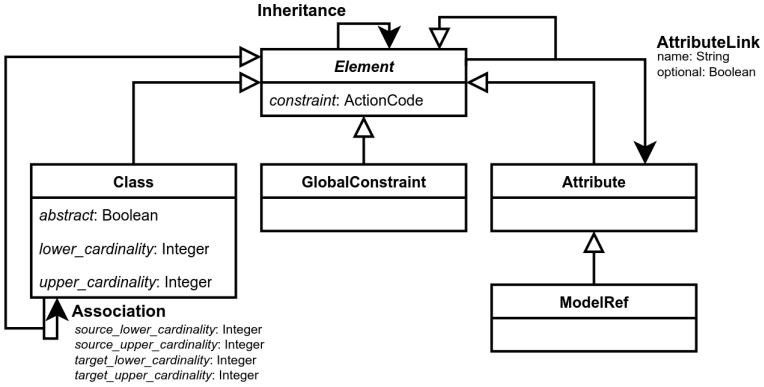

Figure 2: A class diagram conforming to itself (meta-circular).
Conformance-links not shown.
Figure from Andrei Bondarenko's thesis.

2 Overview of Assignment

2.1 Getting Started with muMLE

- Use git to clone the repo `https://github.com/joeriexelmans/muMLE`

  Note: It is recommended to clone the repository and not only download the files. We will use muMLE in future assignments and this allows you to pull potential bug fixes.

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

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

- It is recommended to create a virtual environment (venv) for the project (See `https://www.w3schools.com/python/python_virtualenv.asp` if you have not used a venv before). Install the required packages in your venv using: `pip install -r requirements.txt`.

- You can refer to the `tutorial` directory for help on getting started with muMLE. For this assignment, the relevant tutorials are `00_metamodeling.py`, `01_constraints.py` and `02_inheritance.py`.

2.2 Tasks

To complete the assignment, you will:

1. Create a new file `assignment1.py`. **Add a new meta-model** following the specifications in section 3.
2. Create **two models**:
   - One **conforming** model
   - One **non-conforming** model (include a list of conformance-errors in your report)

   For each of the models, render a visualization in PlantUML.

Write a short PDF report explaining your solution by showing code fragments, your generated PlantUML figures, and your thought process behind everything. Include both team members' names on the report.

2.3 Practical

- Students work in pairs.
- One team member submits a ZIP file containing your report and code (`assignment1.py`).
- Deadline: 15 October 2025, 23:59.

3 Specification

After having worked on character creation in the last assignment, this domain-specific language (DSL) will model the role-playing game (RPG) itself.

- There is exactly one `Hero`, who
  - has a non-negative number of `lives`.
  - is on a `Tile`.
- There is exactly one `World`, which has at least one `Level`.
- There are `Levels`, which have
  - a `name`.
  - at least one `Tile`.
- There are `Tiles`, which
  - have at most 4 adjacent `Tiles` (= top, down, left, right) in the same `Level`.
  - are either a `StandardTile`, `Trap`, `Door` or `Obstacle`.
- A `StandardTile` can have a `Item` on it.
- A `Door`
  - has a `Key`.
  - is connected to a `Door` in a different `Level`.
- An `Item` can be a `Key` or an `Objective`.
- A `Key` belongs to a `Door`.
- An `Objective` has `points`. All `Objectives` together have at most 100 `points` in total, which also means one `Objective` has at most 100 `points`.

4 Constraint API

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

	Availability in Context			
	Local	**Global**	**OD-API**	**Meaning**
Querying				
`this :obj`	✓			Current object or link
`get_name(:obj) :str`	✓	✓	✓	Get name of object or link
`get(name:str) :obj`	✓	✓	✓	Get object or link by name (inverse of `get_name`)
`get_type(:obj) :obj`	✓	✓	✓	Get type of object or link
`get_type_name(:obj) :str`	✓	✓	✓	Same as `get_name(get_type(...))`
`is_instance(:obj, type_name:str [,include_subtypes:bool=True]) :bool`	✓	✓	✓	Is object instance of given type (or subtype thereof)?
`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_slot_value(:obj, attr_name:str) :int|str|bool`	✓	✓	✓	Same as `get_value(get_slot(...))`
`get_all_instances(type_name:str [,include_subtypes:bool=True] ) :list<(str, obj)>`	✓	✓	✓	Get list of tuples (name, object) of given type (and its subtypes).
`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
`has_slot(:obj, attr_name:str) :bool`	✓	✓	✓	Does object have given slot?

(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_slot_value(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)
  types.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"))
```

5 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.