Assignment 2 - Meta-modeling and conformance checking

Stein Vandenbroeke - Ariana Fernández

October 15, 2025

1 Introduction

In this assignment, we create a meta-model using meMLE. The primary difference between Refinery and meMLE is that Refinery automatically generates models from the meta-model, whereas with meMLE we have to create our own models and then can automatically check whether they are conform. meMLE uses Python, and thus it is more intuitive than Refinery for a person who is already familiar with Python to create the constraints.
In the following sections, we describe our design choices, implementation, and the conformance results.

2 Meta-model

2.1 Classes

To design our meta-model, we first identified the necessary classes by carefully reading each specification sentence in our assignment.

For instance, from the sentence "There is exactly one Hero", we immediately concluded that Hero should be a class because it represents an entity (possesses its own properties and participates in relationships with other elements) in our domain. We followed this way of thinking with Level, World, Tile, Item.

To give one example, we are going to show how we implemented Hero class.

```
Hero:Class {
        lower_cardinality = 1;
        upper_cardinality = 1;
        constraint = ```
            get_slot_value(this, "lives") >= 0
        ```;
    }
```

To define the class, first we need to give it a name and declare it using the `:Class` keyword. Inside the curly braces, we define the class cardinality by specifying both a lower and an upper bound, both equal to 1. We can also include internal constraints (we write `constraint = ```...```;` to describe the rule) that ensure something about the entity, for example, in this case, we ensure that the number of lives is non-negative.
Also, I would like to point out that we are using `get_slot_value`, which receives `(:obj, attr name:str)` and returns `:int|str|bool`. In this particular case, it receives an object (Hero), its attribute name (lives) and it returns an integer. That's why we can make this comparison. Note that cardinality tells us how many times a class can participate in a relationship, while constraints help us by describing the logical rules that each instance must follow.

### 2.2 Abstract classes and Inheritance

Once we had identified our classes, we analyzed which of them should be abstract. For example, this is the case for Tile because "There are Tiles, which are either a StandardTile, Trap, Door or Obstacle". This indicates that there are different types of tiles, implying that Tile is a general category and should not be instantiated directly.
```

```
Tile:Class{
        abstract=True;
    }
```

As we can see, we define a class as an abstract class by setting the attribute `abstract` as `true`.

Now, let's talk about inheritance using the same example. The classes StandardTile, Trap, Door, and Obstacle must inherit from Tile, following the hierarchy described in the specification.

To define this inheritance, we do the following:

```
StandardTile:Class
:Inheritance (StandardTile -> Tile)
```

First, we define the class by giving it a name and declaring it with the `:Class` keyword. Then, we use the `:Inheritance(source -> target)` keyword to describe the hierarchy. In this case, StandardTile is the child class and Tile is the parent class.

2.3 Attributes

We identified attributes based on the specification sentences. For example, "There are Levels, which have ... a name" tells us that the class Level needs an attribute called name, and that it is mandatory. So we declare:

```
Level_name:AttributeLink (Level -> String) {
    name = "name";
    optional = False;
}
```

From the example above, we define the attribute by giving it a name and declaring it with the `:AttributeLink` keyword. Then, we use `(Level -> String)` to describe the connection between the class (Level) and the data type of the attribute (String). Finally, we set the properties of the attribute, such as the attribute name and whether it is optional. In this case, we set `optional = False` because every Level must have a name.

This same logic was used to define other attributes, such as points in Objective and lives in Hero.

2.4 Association

Now, we can talk about associations (also called relationships) between classes. The specification sentences also provide direct hints, such as "A Key belongs to a Door." In our framework, we express these relationships using links between classes.

For example, we define an association by giving it a name and declaring it with the `:Association` keyword. Then, we use `(class -> class)` to describe the direction of the relationship we are creating. This relationship can also include cardinalities and constraints if needed. It is worth saying that associations can include both cardinality and optional constraints.

An example of this is shown below:

```
DoorToKey:Association (Door -> Key) {
    target_lower_cardinality = 1;
    target_upper_cardinality = 1;
    source_upper_cardinality = 1;
    source_lower_cardinality = 1;
}
```

In this case, we define that each Door must have exactly one Key, since both the target -lower and -upper cardinalities are set to 1. And that every key has only one door, as determined by the source cardinalities.

2.5 Global Constraints

To conclude with the definition of our meta-model we also define global constraints that apply to all instances in the model. Unlike class or association constraints, global constraints apply to all instances in the model. For example, the constraint `AllObjectivesPointsUnder100` ensures that the total sum of points for all Objectives does not exceed 100.

```
AllObjectivesPointsUnder100:GlobalConstraint {
    constraint = ```
        total_amount_of_objective_points = 0
        for _, objective in get_all_instances("Objective"):
            total_amount_of_objective_points += get_slot_value(objective, "points")

        total_amount_of_objective_points <= 100
    ```;
}
```

We define a global constraint by giving it a name and declaring it with the `:GlobalConstraint` keyword. Then inside the curly braces, we write `constraint = ```...```;` to describe the rule. The code inside the triple backticks is mostly written in Python, which makes it easier to express logical conditions and loops.

This kind of rule allows us to verify conditions that depend on the model as a whole rather than on a single class or relationship.

## 2.6 Other important annotations

The following parts of the code represent constraints that are worth mentioning.

We start by explaining the rule: "There are Tiles, which have at most 4 adjacent Tiles (= top, down, left, right) in the same Level."

To fulfil this constraint, we did the following. Using the associations defined below, we limited both the number of connections between Levels and Tiles, and the number of adjacencies between Tiles.

```
LevelToTile:Association (Level -> Tile) {
    target_lower_cardinality = 1;
    source_upper_cardinality = 1;
}

TileToTile:Association (Tile -> Tile) {
    target_upper_cardinality = 4;
}
```

The first association ensures that each Tile belongs to exactly one Level (this seemed logical, although it was not explicitly stated in the exercise; reusing tiles between levels wouldn't make any sense), while the second one restricts the number of possible Tile-to-Tile connections to a maximum of four. Together, these associations correctly represent the idea that each Tile can have up to four adjacent Tiles in the same Level. The four locations (top, down, left, right) can be given using an attribute `direction` on the TileToTile association (we didn't check whether the top, down, left and right are correctly positioned because this was not explicitly stated in the specification, and because we are making a game it could have some weird dimension rules where going back the same path where you are coming from does not end in the same place).

Now, we are going to explain the rule: "A Door is connected to a Door in a different Level."

To represent this rule, we used the following association and constraint. With this code, we make sure that every Door is linked to exactly one other Door, and that the two connected Doors belong to different Levels.

```
DoorToDoor:Association (Door -> Door){
    target_lower_cardinality = 1;
    target_upper_cardinality = 1;
```
```

```
        constraint = ```
            door0 = get_source(this)
            door1 = get_target(this)

            DoorLevel0 = get_incoming(door0, "LevelToTile")[0]
            DoorLevel1 = get_incoming(door1, "LevelToTile")[0]
            DoorLevel0 != DoorLevel1
            ```;
    }
```

The first part of the association defines that each Door must be connected to exactly one other Door, since both the lower and upper cardinality are set to 1. The constraint inside the association checks that the two connected Doors don't belong to the same Level. To do this, we first get the source and target Doors, and then we identify the Level that each Door belongs to through the `LevelToTile` association. Finally, the condition `DoorLevel0 != DoorLevel1` ensures that those two Levels are different.

# 3 Models

As an introduction to this section, we may first explain what conforming and non-conforming models are. The best way to describe a conforming model is that it follows the structure and constraints of its meta-model. On the other hand, non-conforming model does the opposite, that is, it doesn't fulfil one or more of the rules established in the meta-model.

## 3.1 Conforming model

In this first example, each element in the model corresponds to a class in the meta-model, and all constraints, cardinalities and inheritance relations are met.

There is exactly one instance of `World`, and once instance of `Hero`, which meets the class cardinality (`lower_cardinality = 1`, `upper_cardinality = 1`). This instance also includes the attribute `lives = 10`, a non-negative integer that fulfils the constraint `get_slot_value(this, "lives") >= 0`.

Then we have two instances of `Level`, each with its `name` (`"level1"` and `"level2"`). This follows the rule that every Level must have a mandatory name (`optional = False`). Both Levels are connected to the World through `WorldToLevel`, which fulfils the rule `target_lower_cardinality = 1`, meaning each World must be connected with at least one Level.

Each Level is connected to multiple Tiles through `LevelToTile`, and every Tile belongs to exactly one Level. The Tiles (`Trap`, `StandardTile`, `Obstacle`, `Door`) follow the inheritance hierarchy, since they all inherit from the abstract class `Tile`.

When we check the constraints, each `Door` has exactly one `Key` (`len(get_incoming(this, "DoorToKey")) == 1`). Also, `DoorToDoor` connects doors from different levels, respecting the global rule (`DoorLevel0 != DoorLevel1`).

Another rule is the global constraint `AllObjectivesPointsUnder100`, which ensures that the total points from all objectives don't exceed 100. Here, two instances of `Objective` have `points = 50`, so the total is 100 and the constraint is fulfilled.

Additionally, the directional connections between Tiles are defined by `TileToTile`. For example, `T5_T6 (T5 -> T6)` has `direction = "up"`, and others like `"down"`, `"left"` or `"right"`. This ensures that every link has a valid direction and that the attribute `direction` (mandatory) is always defined.

So, as all instances, attributes and relations in this model follow the meta-model rules, it can be considered a conforming model.

```
comform_m = """
    W:World

    H:Hero{
        lives = 10;
    }
```
```

```
 L1:Level{
     name = "level1";
 }

 L2:Level{
     name = "level2";
 }

 W_L1:WorldToLevel (W -> L1)
 W_L2:WorldToLevel (W -> L2)

T1:Trap
T2:StandardTile
T3:StandardTile
T4:Obstacle
D0:Door
D1:Door

D0_D1:DoorToDoor (D0 -> D1)
D0_D2:DoorToDoor (D1 -> D0)

L1_T1:LevelToTile (L1 -> T1)
L1_T2:LevelToTile (L1 -> T2)
L1_T3:LevelToTile (L1 -> T3)
L1_T4:LevelToTile (L1 -> T4)
L1_T5:LevelToTile (L1 -> D0)

T1_T2:TileToTile (T1 -> T2){
 direction = "up";
}

T5:StandardTile
T6:StandardTile
T7:StandardTile

L2_T5:LevelToTile (L2 -> T5)
L2_T6:LevelToTile (L2 -> T6)
L2_T7:LevelToTile (L2 -> T7)
L2_D1:LevelToTile (L2 -> D1)

T5_T6:TileToTile (T5 -> T6){
 direction = "up";
}

T6_T5:TileToTile (T6 -> T5){
 direction = "down";
}

T5_T7:TileToTile (T5 -> T7){
 direction = "left";
}

T7_T5:TileToTile (T7 -> T5){
 direction = "right";
}
```

```
    H_T0:HeroTile (H -> T1)

    K0:Key
    K1:Key
    T2_K0:StandardToTileItem (T2 -> K0)
    T3_K1:StandardToTileItem (T3 -> K1)
    D0_K0:DoorToKey (D0 -> K0)
    D1_K1:DoorToKey (D1 -> K1)

    O1:Objective{
        points = 50;
    }


    O2:Objective{
        points = 50;
    }
"""
```

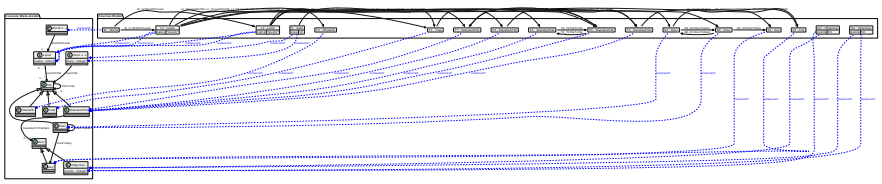

Figure 1: Conforming model

The diagram helps us to visualise this conformance. On the left, we see the classes and relations defined in the meta-model, while on the right, we see specific instances that conform to those classes.

The lines connecting them "conformance links" show that each element in the model has a valid correspondence with its definition in the meta-model. This visual representation confirms that our model fully satisfies the meta-model and that all constraints are respected.

4 Non-conforming model (include a list of conformance-errors in your report)

Compared to the previous model, this version intentionally breaks several structural and logical rules defined in the meta-model. We aim to demonstrate the consequences of not respecting certain constraints and cardinalities, resulting in a non-conforming model.

```
nonconform_m = """
    W:World

    H:Hero{
        lives = 10;
    }

    L1:Level{
        name = "level1";
    }

    L2:Level{
```

```
        name = "level2";
    }

    W_L1:WorldToLevel (W -> L1)
    W_L2:WorldToLevel (W -> L2)

    T1:Trap
    T2:StandardTile
    T3:StandardTile
    T4:Obstacle

    T1_T2:TileToTile (T1 -> T2){
        direction = "right";
    }

    T3_T2:TileToTile (T3 -> T2){
        direction = "right";
    }

    D0:Door
    D1:Door

    D0_D1:DoorToDoor (D0 -> D1)
    D1_D0:DoorToDoor (D1 -> D0)

    L1_T1:LevelToTile (L1 -> T1)
    L1_T2:LevelToTile (L1 -> T2)
    L1_T3:LevelToTile (L1 -> T3)
    L1_T4:LevelToTile (L1 -> T4)
    L1_D0:LevelToTile (L1 -> D0)
    L1_D1:LevelToTile (L1 -> D1)

    L2_T1:LevelToTile (L2 -> T1)
    L2_T2:LevelToTile (L2 -> T2)
    L2_T3:LevelToTile (L2 -> T3)
    L2_T4:LevelToTile (L2 -> T4)

    H_T0:HeroTile (H -> T1)
    H_T1:HeroTile (H -> T2)

    K:Key
    T2_K:StandardToTileItem (T2 -> K)
    D0_K:DoorToKey (D0 -> K)
    D1_K:DoorToKey (D1 -> K)

    O1:Objective{
        points = 60;
    }

    O2:Objective{
        points = 60;
    }
"""
```

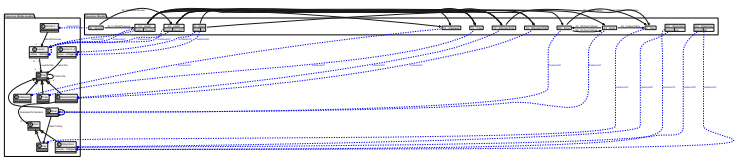

Figure 2: Non-conforming model

When this model was validated (Figure 3), it was correctly marked as non-conforming. The validation output shows several errors caused we breaking some cardinality rules and constraints, such as:

The source cardinality of `LevelToTile` is out of bounds, because several Tiles are linked to the same Level, but only one connection is allowed (0..1). The target cardinality of `HeroTile` is also broke, since the Hero is connected to two Tiles instead of just one. The local constraint of `Key` is not satisfied because there are two tiles on the right of T2. The local constraint of `Key` isn't met, because one Key is connected to two doors instead of exactly one. The global constraint `AllObjectivesPointsUnder100` fails, because both objectives together add up to 120 points, which goes over the allowed total of 100.

These issues show how muMLE checks both the structure of the model (its cardinalities and associations) and its logic (local and global constraints). By adding these intentional mistakes, this model clearly shows how muMLE detects and reports when a model does not conform to its meta-model.

```
Load SCD
Done
Parsing MM
Done
Parsing Model (Conforming model)
Done
Parsing Model (Non-conforming model)
Done
Valid? (Conforming model)
{'up'}
{'up'}
{'left'}
{'down'}
{'right', 'down'}
CONFORM
Valid? (Non-conforming model)
{'right'}
{'right'}
NOT CONFORM, 9 errors:
  ‣ Source cardinality of type 'DoorToKey' (2) out of bounds (1..1) in 'K'.
  ‣ Source cardinality of type 'LevelToTile' (2) out of bounds (0..1) in 'T4'.
  ‣ Source cardinality of type 'LevelToTile' (2) out of bounds (0..1) in 'T2'.
  ‣ Source cardinality of type 'LevelToTile' (2) out of bounds (0..1) in 'T3'.
  ‣ Source cardinality of type 'LevelToTile' (2) out of bounds (0..1) in 'T1'.
  ‣ Target cardinality of type 'HeroTile' (2) out of bounds (1..1) in 'H'.
  ‣ Local constraint of "Tile" in "T2" not satisfied.
  ‣ Local constraint of "Key" in "K" not satisfied.
  ‣ Global constraint "AllObjectivesPointsUnder100" not satisfied.
```

Figure 3: Terminal Output

5 Python code

Last but not least, our TA provided a script that loads the meta-model and the model, checks whether the model conforms to the meta-model, and then generates a UML diagram and a visual link with the

results. We used it to validate both our conforming and non-conforming models.