MDE Assignment 6:
Translational Semantics - Rule-Based Model Transformation

Joeri Exelmans
contact: joeri.exelmans@uantwerpen.be

December 4, 2024

1 Introduction

In this assignment, we will implement the same semantics of our 'port'-DSL, that we are already familiar with from previous assignments, but this time, we will do so by transforming our DSL to Petri Nets.

The benefit of transforming to another formalism, is that the entire language toolchain (in the case of Petri Nets: interactive simulation and highly optimized analysis/model checking) becomes available to us.

The translation will be done by means of rule-based model transformation, which we are already familiar with.

2 Translational Semantics in muMLE

Model transformation in muMLE always happens within the same formalism. To transform *across* formalisms, we perform a simple trick of merging the meta-models of the input and output formalisms. In our case, this means merging the meta-models of the 'Port'-DSL and the 'Petri Net'-language.

When merging (meta-)models, naming collisions can occur. In this assignment, the types of both formalisms have been named specifically to avoid naming collisions.

Once we have our merged meta-model, we modify it in one more way: We add an abstract class named `Top`, which becomes the superclass of all classes[1]. We also add an association named `generic_link`, whose source and target is `Top`. This effectively allows `generic_link` to connect anything with anything.

Did you know: adding the `Top`- and `generic_link`-types, and the necessary inheritance-links, is also implemented with model transformation!

The merged meta-model (Figure 1) can be found in `examples/semantics/translational/merged_mm.od`. It was generated by the script `examples/semantics/translational/regenerate_mm.py`. Note that the merged meta-model is also a valid meta-model for our existing 'Port'-language: it merely contains additional

[1] In type theory, the 'top'-type is the supertype of all types. See `https://en.wikipedia.org/wiki/Top_type`

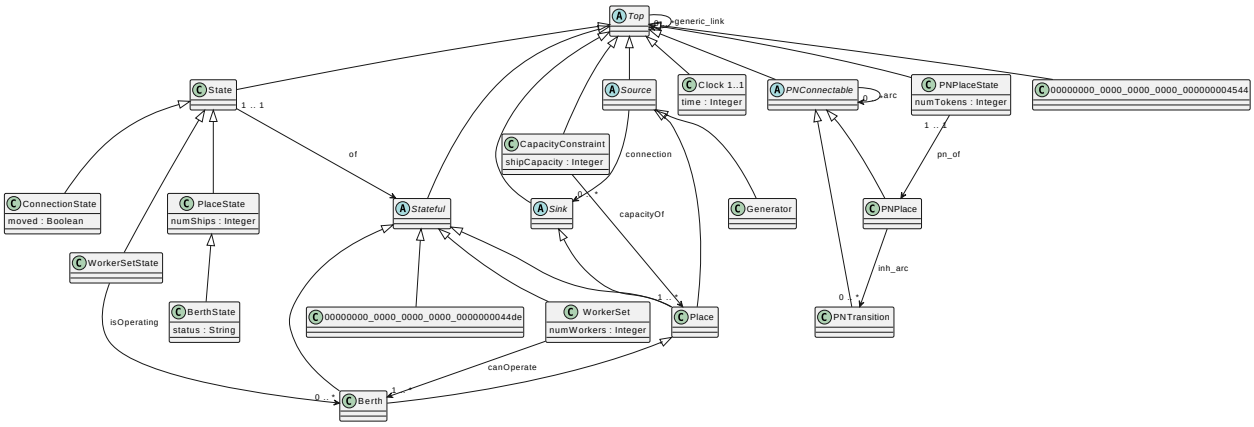

Figure 1: Our merged meta-model: Port and Petri Nets

types. In the general case, adding new types to a meta-model does not break compatibility with existing models.

Next, the merged meta-model is RAMified, and we can finally create model transformation rules that transform 'Port'-models into 'Petri Net'-models. We will do this not in a single rule, but with a multiple rules, which build the Petri Net model incrementally. We will also not erase the 'Port'-model (or any of its elements), but rather "append" the Petri Net model to it, and we will create traceability-links from the created 'Petri Net'-elements back to their respective 'Port'-elements.

2.1 Example: Translate Port-Place into Petri Net-Place

The following LHS will look for a Place and its PlaceState (from the 'Port'-language)

Listing 1: LHS

```
port_place:RAM_Place
port_place_state:RAM_PlaceState
port_of:RAM_of (port_place_state -> port_place)
```

The RHS will create a Petri Net-Place with the same number of tokens as the number of ships in the Port-PlaceState:

Listing 2: RHS

```
# We repeat the entire LHS:
port_place:RAM_Place
port_place_state:RAM_PlaceState
port_of:RAM_of (port_place_state -> port_place)

# To create: a Petri Net-place, and link it to our Port-place
pn_place:RAM_PNPlace {
  # new feature: you can control the name of the object to be created
  # using names based on the Port-elements makes the translation much more understandable!
  name = 'f"ships_{get_name(matched("port_place"))}"';
}
place2place:RAM_generic_link (pn_place -> port_place)
```

```
# Also create: a Petri Net-PlaceState (indicating the amount of tokens in our newly created place)
pn_place_state:RAM_PNPlaceState {
  RAM_numTokens = ‘get_slot_value(matched(’port_place_state’), "numShips")‘;
}
:RAM_pn_of(pn_place_state -> pn_place)
```

We also define a NAC, to prevent the rule from firing more than once for every match. In this case, the NAC simply consists of all the elements from the RHS:

Listing 3: NAC

```
port_place:RAM_Place
port_place_state:RAM_PlaceState
port_of:RAM_of (port_place_state -> port_place)
pn_place:RAM_PNPlace
place2place:RAM_generic_link (pn_place -> port_place)
pn_place_state:RAM_PNPlaceState
:RAM_pn_of(pn_place_state -> pn_place)
```

This rule, and another rule translating a Port-connection into a Petri Net-transition are included in the starting point for this assignment.

2.2 Specification of Semantics

Note: this specification has not changed since the previous assignment.

- A ship can move along a connection, only:
  - if there is at least one ship in the source of the connection, with some exceptions:
    - A Generator can be considered a special kind of source, always having ships available.
    - A ship can only leave a Berth if it has been served.
  - if there is enough capacity in the target/sink of the connection.
  - if no ship has moved yet over the connection, during the current time step.
- Further, a connection only becomes ‘active’ if all connections after it have had a chance to make a move.
  - For instance, in **??**, the connection ‘outboundPassage’ → ‘served’ occurs after ‘outboundBerth2’ → ‘outboundPassage’. The former will thus have priority.
- Along an ‘active’ connection, if a ship can move, it must move, and otherwise, the connection is skipped (marking it as ‘moved’ without having moved a ship).
- If a ship is at a Berth, and the status of the Berth is “unserved”, a worker may be assigned to the Berth, but only if:
  - There is a ‘canOperate’-link from the WorkerSet to the Berth
  - The WorkerSet still has a worker available. In other words, the number of outgoing ‘isOperating’-links must be smaller than the size (‘numWorkers’) of the WorkerSet.
- If none of the above actions are possible anymore, a time step ends, having the following effects:

- The current time is incremented.
- For every worker that is operating a Berth, the Berth's status changes to "served", and the worker stops operating the Berth. (In the next time step, the worker can be assigned again to a Berth)
- The 'moved' flag of every connection is reset to False.

3 Getting Started

- Do a `git pull` to get the latest sources.
- Under `examples/semantics/translational`, you'll see the following files:
  - Files you should not edit:
    - `merged_mm.od` The merged meta-model.
    - `regenerate_mm.od` The script that was used to generate the merged meta-model.
    - `runner_exec_pn.od` Once you have generated a Petri Net, you can execute it with this script.
  - Files you should edit:
    - `runner_translate.py` This is the script that will perform the Port-to-Petri Net translation.
    - `rules/gen_pn` In this directory, you will place your translation rules. There are already two rules here, which you can (but don't have to) use as a starting point for your solution.

The script `runner_translate.py` that will carry out the translation, will execute the rules in a specific manner. Rules are specified in an explicit order of priorities. It will first try to find a match for the highest-priority rule, and execute that rule for the first match that was found. Then, it will again try to find a match for the highest-priority rule, and so on. Only if no match is found, will the next rule be attempted. This is repeated until all rules have been attempted. You can construct your rules such that the Petri Net is generated in an iterative manner.

Executing the translation rules can take some time, especially as the model grows large. In order not to run the entire transformation from scratch, the `runner_translate.py` script will save a snapshot of the ongoing translation every time a rule has been "exhausted" (it has fired as much as it could). When you re-run the script, the script will re-load existing snapshots rather than re-executing rules. For instance, if your translation consists of 8 rules, and you modify the last rule, you don't want to re-execute the first 7 rules. To force rule execution, simply delete the snapshot-files.

To execute a generated Petri Net, simply run `runner_exec_pn.od` with the model file (e.g., the most recently created snapshot) as a parameter.

4 Tips

- You can work incrementally:
  - for instance, first generate objects, then links, in (many) separate rules.

  - more concretely, you can first create Petri Net places that store the remaining capacity of a CapacityConstraint, and later connect those places to the correct Petri Net transitions.

- When working incrementally, you can match traceability links (of type `generic_link`) that were created by earlier rules, in the LHS of a later rule.
- You'll have to think of a way to encode the entire Port-run-time state into a Petri Net. For instance, how would you encode the status of a berth (served/unserved)? You'll need to think in advance about all these things.
- Work out the transformation rules visually on a piece of paper, before writing them down in muMLE. Otherwise it is easy to get lost.
- When creating an object or link in the RHS of a rule, you can specify its name with the `name`-attribute, as demonstrated in Listing 2. By making the names of Petri Net elements refer to the names of the Port-elements to which they relate, you can greatly improve the understandability of your generated Petri Net.

> One (dirty, but useful!) example of exploiting the `name`-attribute is the following: Suppose I am creating two places for a Berth named 'b1': one that counts the number of ships in the Berth, another that holds a token if a worker is assigned to it. By naming the places 'ships_b1' and 'worker_b1', I can distinguish between them. In a later rule's LHS, I can even write a condition that checks the name (e.g., using the Python function `.startswith("ships_")`) if I want to match with a specific kind of place. This also has the benefit of speeding up the matching!

- When rendering the Petri Net with GraphViz, try switching engines if you're getting poor results. The 'neato'-engine seems to work quite well for Petri Nets. Also remember that the result does not have to be perfect.
- This will be quite a big assignment: start early!

5 Practical

- Students work individually.
- Submit, via Blackboard, a ZIP file containing:
  - your **rules** directory, containing the transformation rules
  - your **runner_translate.py** file
  - the result of your translation (`.od`-extension)
  - the GraphViz-rendered Petri Net (as a figure)
  - a small report, where you explain the different rules you created.
- Deadline: Tuesday 17 December 2024, 23:59.

6 API

Here is the API, once more:

	Availability in Context					
	Meta-Model Constraint		Model Transformation Rule			
	Local	**Global**	**NAC LHS**	**RHS**	**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?
`matched(label:str) :obj`			✓	✓		Get matched object by its label (the name of the object in the pattern)
Modifying						
`delete(:obj)`				✓	✓	Delete object or link
`set_slot_value(:obj, attr_name:str, val:int|str|bool)`				✓	✓	Set value of slot. Creates slot if it doesn't exist yet.
`create_link(link_name:str|None, assoc_name:str, src:obj, tgt:obj) :link`				✓	✓	Create link (typed by given association). If `link_name` is None, name is auto-generated.
`create_object(object_name:str|None, class_name:str) :obj`				✓	✓	Create object (typed by given class). If `object_name` is None, name is auto-generated.

If there is an API function that you would like to see added, contact me.