MDE Assignment 4: Operational Semantics - Coded in Python

Joeri Exelmans
contact: joeri.exelmans@uantwerpen.be

November 5, 2024

1 Introduction

In previous assignments, we have touched the following topics:

- Meta-modeling
- Instance generation
- Conformance checking
- Concrete and abstract syntax

Of course, a language is not useful without *meaning*, i.e., *semantics*. Depending on the language, the meaning can be pretty much anything. In the case of Class Diagrams, the meaning is a (possibly infinite) set of conforming object diagrams. In the case of executable languages, the meaning is a (possibly infinite) set of execution traces. In this assignment, we will code an *operational semantics* for an executable language.

In model-driven engineering, *operational semantics* is a semantics of the form that evolves an *execution state*, also called *runtime state* or *runtime configuration* from one 'snapshot' to the next. The runtime state may point to parts of the *design model* (the model that was manually created). The design model never changes during execution[1].

2 Assignment

Figure 1 shows an impression of a system that we want to model: Ships are arriving from the open sea, and want to get to a dock, where they will be unloaded ("served"). To get to the dock, they have to go through a narrow passage first, which only has a capacity for 3 ships. Ships leaving from the dock also need to go through the narrow passage (where they may block or be blocked by arriving ships). The dock consists of 2 berths, each of which can hold 1 ship. Each berth has an entry and exit that connects to the passage, also capable of holding 1 ship. There is a single worker on the dock, who can unload 1 ship at a time.

Figure 2 shows a possible way to model this: we have Places (blue rountangles) that can hold ships, we have connections (blue arrows) between Places, and we have Berths (yellow rountangles), which are a special kind of Place. We have sets of workers (green circle), that can serve multiple Berths (purple arrows). Places

[1] Except in *live modeling*.

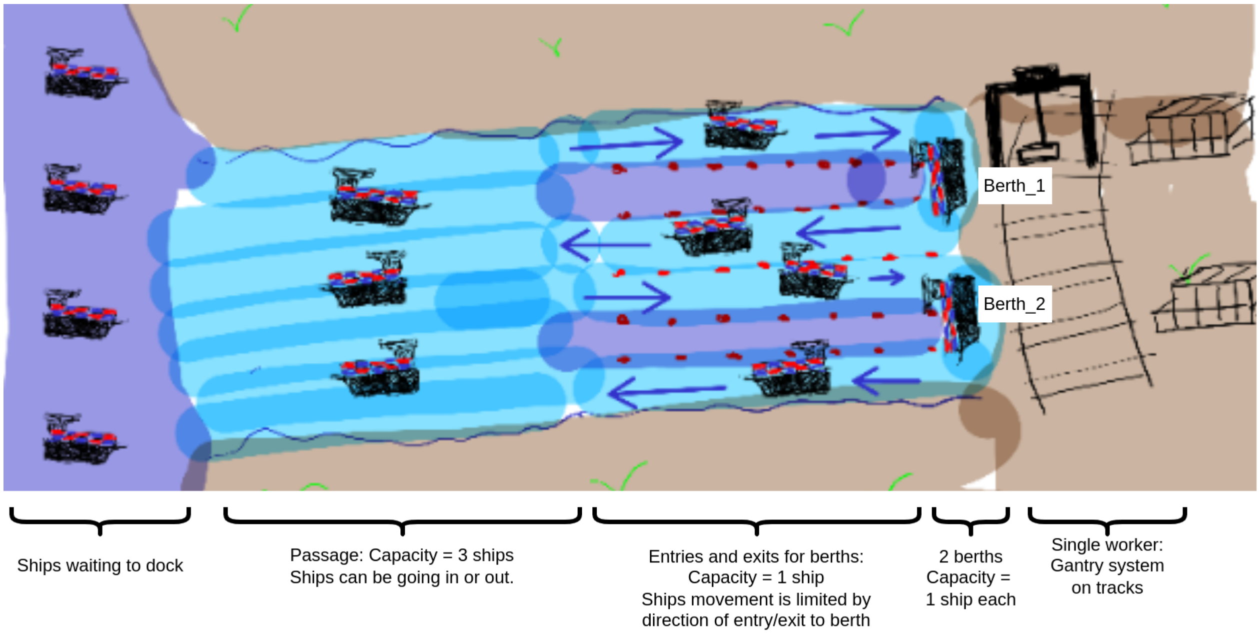

Figure 1: Artistic impression of our system. (credit goes to Rakshit Mittal)

can be capacity-constrained (grey rectangles). The arrival of ships is taken care of by a Generator (orange). Ships that leave are put in a place called 'served', which has infinite capacity. This way, we can count the number of processed ships.

More formally, the top part of Figure 3 shows a meta-model of our design language. The bottom part shows the types that make up our runtime state. We have:

- **PlaceState** keeps track of the number of ships in a Place.
- **BerthState** extends PlaceState, to also keep track of the state of a ship in a Berth (served or unserved).
- **WorkerSetState** keeps track of the Berths that are currently being operated by a set of workers.
- **Clock** counts the number of time-steps.
- **ConnectionState** keeps track of the state of every connection: along every connection, only one ship can move per time-step.

2.1 Specification of Semantics

The precise semantics of our language are as follows:

- 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.

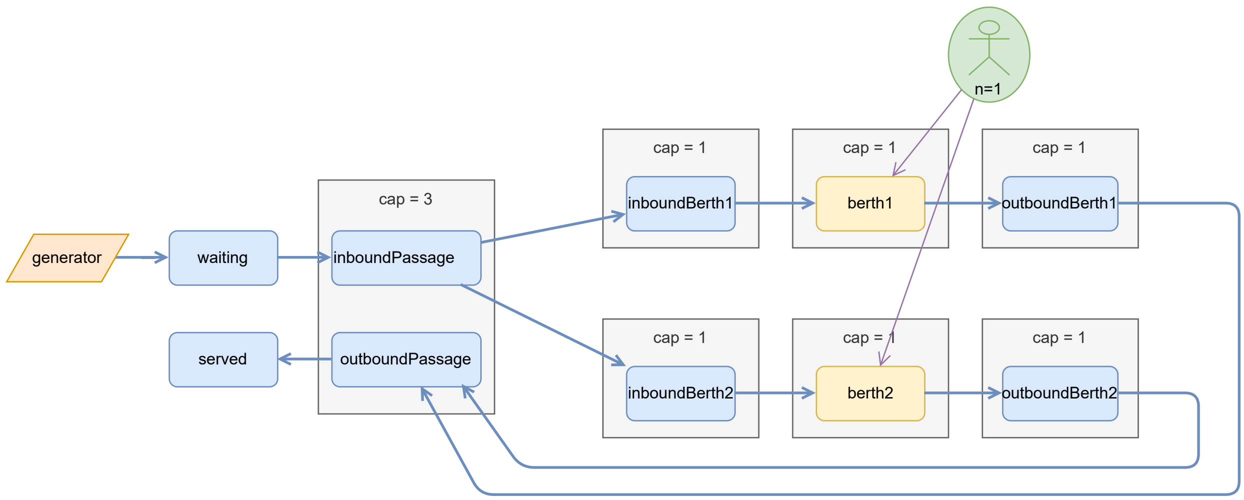

Figure 2: Our (design) model.

    * 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 Figure 2, 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.

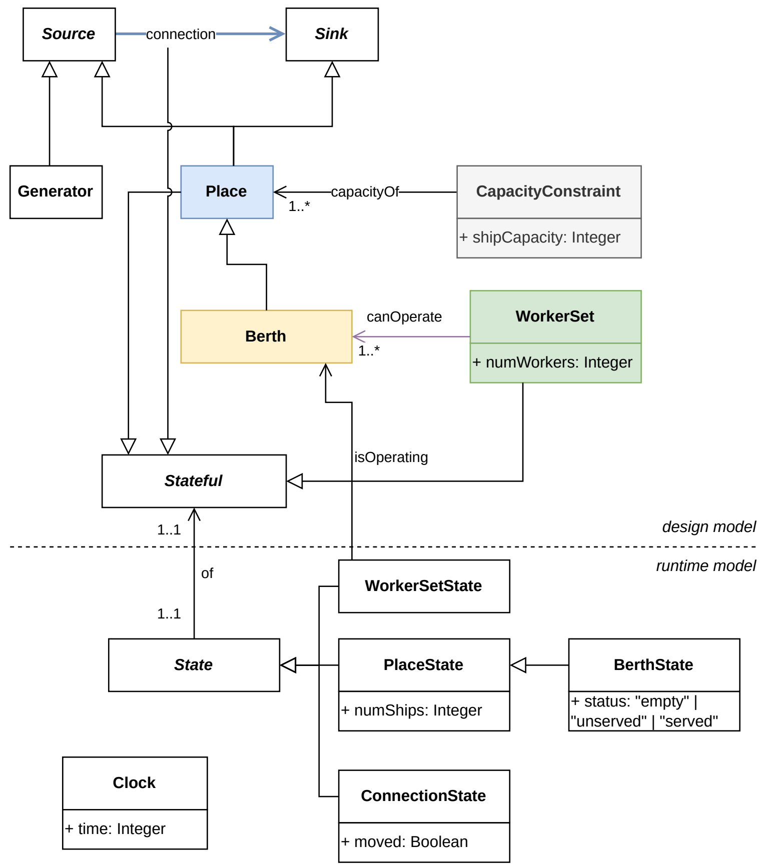

Figure 3: Class diagram of design- and runtime meta-models. Runtime meta-model is a superset of design meta-model.

3 Getting started

- Work continues in the 'muMLE' repository. Do a 'git pull' to get the latest version.
  - `https://msdl.uantwerpen.be/git/projects/muMLE`
  - `https://github.com/joeriexelmans/muMLE` (mirror)
- In the directory `semantics/operational/port`, the following files are of interest:
  - **runner.py** This is the Python script that runs the simulation.
    Hint: You can switch between interactive and automated simulation by changing the `decision_maker` parameter of the Simulator class.
    Hint: You can also switch the renderer (Graphviz or textual).
  - **models.py** This file contains the (meta-)models for design and runtime state.
    Hint: The meta-models contain many constraints that are automatically checked after every execution step. If an execution step makes the runtime state non-conforming, then you've made a mistake!
  - **assignment.py** This is the main file you should edit. Some parts are already implemented. Look for the TODOs.

4 API

In the Python functions, whenever you see an object named `od`, it is an instance of the class `ODAPI` ("Object Diagram API"), defined in `api/od.py`. It extends the query-functions of the API from assignment 2 with methods for creating, modifying and deleting:

	Available in Context			
	Local Constraint	**Global Constraint**	**ODAPI**	**Meaning**
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?
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.

5 Remarks

- In the function `precondition_can_move_from(od, from_state)`, if the parameter `from_state` is `None`, you may assume that it is checking whether a move can be made from a Generator. This is because a generator has no associated runtime state.

6 Practical

- Students work individually.
- Submit, via Blackboard, a ZIP file containing:
  - your **assignment.py** file
  - an execution trace (copy-paste terminal output)
  - for this assignment, *you don't have to write a report.*
- Deadline: Tuesday 12 November 2024, 23:59.

7 Extra material

As an example, the `examples/semantics/operational/woods_runner.py` runs a simulation of an operational semantics for our beloved 'woods' language.