MDE Assignment 5: Operational Semantics - Rule-Based Model Transformation

Joeri Exelmans
contact: joeri.exelmans@uantwerpen.be

November 21, 2024

1 Introduction

In this assignment, we will implement the same operational semantics of the 'port'-DSL from the previous assignment, but this time, using rule-based model transformations. During simulation, there is still a *design model* (that doesn't change), and a *runtime model* (that changes after every execution step). The only difference is how the execution steps are defined: namely as a set of *transformation rules*, as opposed to Python code.

A transformation rule classically contains three parts:

Left-Hand Side (LHS) A pattern that must occur, for the rule to produce a match.

Negative Application Condition (NAC) A pattern that must not occur, for the rule to produce a match.

Right-Hand Side (RHS) Describes what the matched pattern should look like *after* executing the rule.

LHS and NAC together define when a rule can fire. First, the LHS-pattern is searched for in the model that is being transformed (this model is sometimes called the *host graph*). For every LHS-match, the matcher attempts to *grow* the match with the elements defined in the NAC. Only if the latter does *not* succeed, does the entire rule match.

LHS and RHS together define what happens when the rule fires: Any element that occurs in the LHS, but not in the RHS, is deleted. Any element that does not occur in the LHS, but occurs in the RHS, is created. Any element that occurs in both LHS and RHS is left untouched (with the exception of attributes, which can be updated, as we will see).

The LHS-, NAC- and RHS-patterns that a transformation rule consists of, are models too: they are instances of the *RAMified runtime model*. RAMification turns the meta-model of the host graph into a new, slightly different meta-model: for instance:

- the lower-cardinalities for all types become zero
- abstract classes become concrete
- the types of attributes are turned into ActionCode (LHS/NAC: boolean expression indicating whether to match, RHS: expression of new value)
- every class gets an additional `condition`-attribute

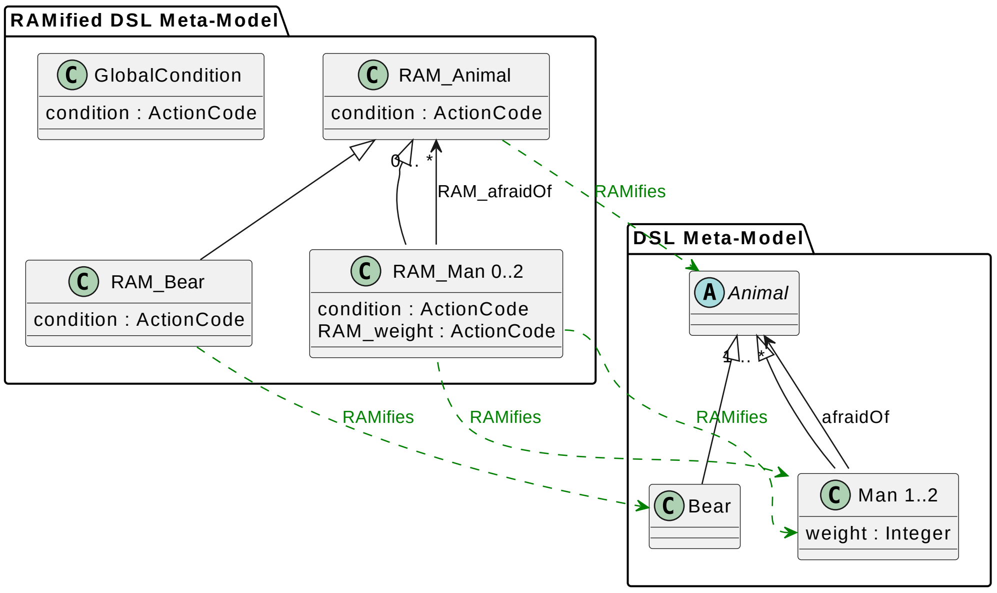

Figure 1: RAMification of our ‘woods’-metamodel.

- a `GlobalCondition`-class is added

RAMification can be done automatically. Figure 1 shows an auto-generated RAMified meta-model of our ‘woods’-formalism.

2 Assignment

See previous assignment for introduction to the Port-DSL.

2.1 Specification of Semantics

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

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.
    - 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.

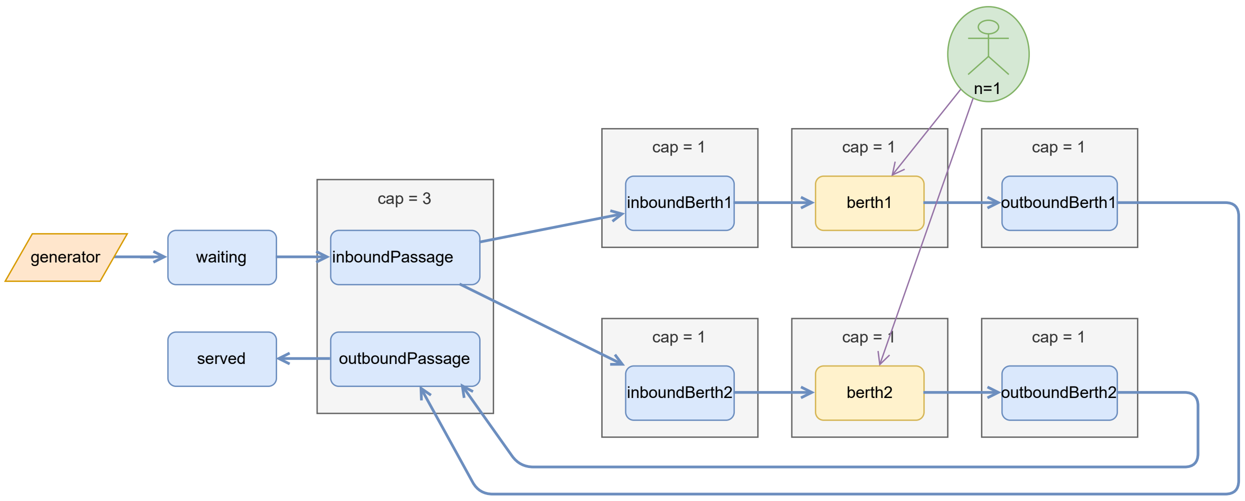

Figure 2: Our (design) model.

  - 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.

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:
  - Files you should not edit:
    * **rulebased_runner.py** The main runner. Almost identical to `runner.py`, but instead derives actions from rules defined in `rulebased_sem.py`.
      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!
      Hint: For testing, you can temporarily alter the initial runtime model, to start simulation from a different state (e.g., one that already has ships in some places).
  - Files you should edit:
    * **rules/** In this directory, you will put your model transformations.
    * **rulebased_sem.py** In this file, you will load your model transformations, and possibly define a priority between them. Rules of lower priority can only fire when no rules of higher priority can fire. You will also implement a termination condition, in the form of a pattern (not a rule). Look for 'TO IMPLEMENT'.

4 API

The same API that we are already familiar with, is available in the attributes of objects in LHS / NAC / RHS patterns:

	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.

5 Tips

- To update a slot, the slot must occur in both LHS and RHS. If the slot only occurs in RHS, it will be created (possibly leading to an object having two attributes with the same name).
- Items in NAC/LHS/RHS are matched by their *name* in the pattern. Therefore, it is best to not use anonymous objects/links in patterns.
- Multiple correct solutions exist. The RHS of every rule can contain arbitrary code (in a `GlobalCondition`). Code is needed sometimes because model transformation rules by themselves are less expressive than code[1]. However, try to use not too much code! Solutions that are easy to explain / understand are favored.
- NACs are optional. If needed, a rule can have more than one NAC. To specify more than one NAC, the NACs must be named `r_rule_nac.od`, `r_rule_nac2.od`, `r_rule_nac3.od`, ...

[1]For this reason, extensions, such as amalgamated rules, have been suggested.

- Beware that an empty NAC (i.e., a NAC that exists as a file, but contains no elements) will *always* match, therefore causing the entire rule to *never* match.

6 Practical

- Students work individually.
- Submit, via Blackboard, a ZIP file containing:
  - your **rules** directory, containing the transformation rules
  - your **rulebased_sem.py** file
  - an execution trace (copy-paste terminal output)
  - a small report, where you explain the different rules you created, and the termination condition pattern.
- Deadline: Tuesday 26 November 2024, 23:59.

7 Extra material

Examples of model transformation in muMLE:

- The `examples/woods/opsem_rulebased.py` implements the semantics of our 'woods'-formalism as a bunch of transformation rules.
- In `examples/cbd`, you can find an implementation of a minimal Causal Block Diagrams (CBD) language, and its semantics, with transformation rules. It includes an example model that computes Fibonacci numbers.
- In `examples/petrinet`, you'll find a Petri Net language, its semantics implemented with (only 1) model transformation rule.
- Note that the distinction between *runtime model* and *design model* is not necessary for model transformation: *all meta-models* can be RAMified, and their instances transformed. For instance, Figure 1 shows the RAMification of the design model of the woods-formalism.