



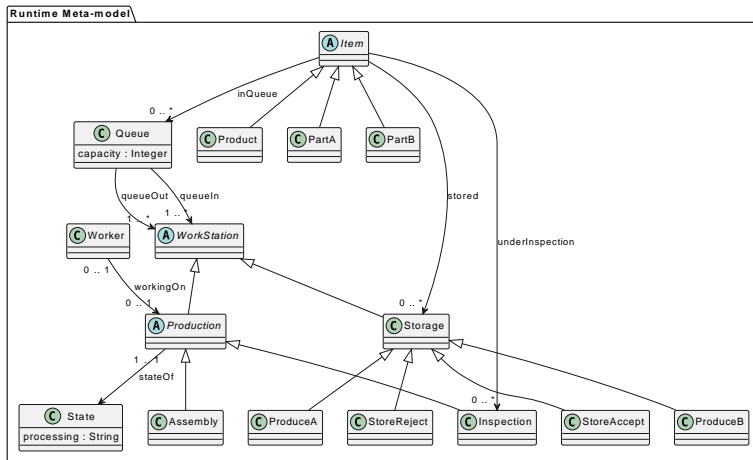
University of Antwerp
| Faculty of Science

Rule-based Operational Semantics - Demo

Vanessa Flügel

November 2025

Factory Example: (Runtime) Meta-Model



Defining a Rule: LHS

```
assemblyStation:RAM_Assembly
assemblyState:RAM_State {
    RAM_processing = 'get_value(this) == "ready"';
}
isAssemblyState:RAM_stateOf (assemblyStation -> assemblyState)

inA:RAM_Queue
inB:RAM_Queue

aGoesIn:RAM_queueIn (inA -> assemblyStation)
bGoesIn:RAM_queueIn (inB -> assemblyStation)
```

Defining a Rule: LHS

```
a:RAM_PartA
aEnqueued:RAM_inQueue (a -> inA)

b1:RAM_PartB
b1Enqueued:RAM_inQueue (b1-> inB)
b2:RAM_PartB
b2Enqueued:RAM_inQueue (b2 -> inB)

aWorker:RAM_Worker
```

Defining a Rule: RHS

```
assemblyStation:RAM_Assembly
assemblyState:RAM_State {
    RAM_processing = "processing";
}
isAssemblyState:RAM_stateOf (assemblyStation -> assemblyState)

inA:RAM_Queue
inB:RAM_Queue

aGoesIn:RAM_queueIn (inA -> assemblyStation)
bGoesIn:RAM_queueIn (inB -> assemblyStation)

aWorker:RAM_Worker
:RAM_workingOn (aWorker -> assemblyStation)
```

Defining a Rule: NAC

```
aWorker:RAM_Worker
```

```
anotherThing:RAM_Production
```

```
alreadyWorking:RAM_workingOn (aWorker -> anotherThing)
```

Parsing Rules and Creating an Action Generator

```
def get_rules(current_state, rt_mm):
    rt_mm_ramified = ramify(current_state, rt_mm)
    matcher_rewriter = RuleMatcherRewriter(current_state,
                                             rt_mm, rt_mm_ramified)

    rules0 = load_rules(current_state, get_filename,
                        rt_mm_ramified, ['store'])
    [...]
    rules4 = load_rules(current_state, get_filename,
                        rt_mm_ramified, ['add_A', 'add_B'])

    return PriorityActionGenerator(matcher_rewriter, [rules0,
                                                    rules1, rules2, rules3, rules4])
```

Termination Condition: Rules

```
patterns_cs = {  
    "There are at least two items accepted": ""  
        anAccept:RAM_StoreAccept {  
            condition = 'len(get_incoming(this, "stored")) >=  
                2';  
        }  
        "",  
    "There is a rejected item": ""  
        aReject:RAM_StoreReject {  
            condition = 'len(get_incoming(this, "stored")) >=  
                1';  
        }  
        ""  
}
```


Termination Condition: Parse and Check

```
class TerminationCondition:
    def __init__(self, state, rt_mm):
        self.state = state
        self.rt_mm_ramified = ramify(state, rt_mm)

        patterns_cs = { [...] }

        self.patterns = {cause: parse_od(state, pattern_cs,
                                          self.rt_mm_ramified)
                        for cause, pattern_cs in patterns_cs.
                          items()}

    def __call__(self, od):
        for cause in self.patterns:
            for match in match_od(self.state, od.m, od.mm,
                                  self.patterns[cause], self.rt_mm_ramified):
                return cause # Stop after first match
```

Running the Simulation

```
rule_generator = get_rules(state, rt_fact_mm)
rule_sim = FactorySimulator(
    action_generator=rule_generator,
    termination_condition=TerminationCondition(state,
        rt_fact_mm),
    renderer=render_text
)

factory_od = ODAPI(state, rt_fact_m, rt_fact_mm)
rule_sim.run(factory_od)
```

Output

```
🏭 reject (□□□□□) 🏭 --> rejectSink
storeA --> 🏭 inA (⚙️⚙️□□□) 🏭
storeB --> 🏭 inB (⚙️□□□□) 🏭
🏭 inB (⚙️□□□□) inA (⚙️⚙️□□□) 🏭 --> assemblyStation [🔧] --> 🏭 toInspection (□□□) 🏭
🏭 accept (□□□□□) 🏭 --> acceptSink
🏭 toInspection (□□□) 🏭 --> inspectionStation [🔧] --> 🏭 reject (□□□□□) accept (□□□□□) 🏭
```

CONFORM

warning: pattern has multiple components: 2

warning: pattern has multiple components: 2

a. add_A

{'prodA': 'storeA', 'productionQueue': 'storeAOut', 'queueA': 'inA'}

b. add_B

{'prodB': 'storeB', 'productionQueue': 'storeBOut', 'queueB': 'inB'}

Select action: