Assignment 1
Production System Modelling in metaDepth

Bentley James Oakes

1 Practical Information

The goal of this assignment is to design a domain-specific modelling language (formalism) and subsequently to model production systems (a factory) in that language in the textual modelling tool metaDepth.

The different parts of this assignment:

1. Implement the abstract syntax of your language in metaDepth.
2. Enrich the abstract syntax with constraints (using EOL) so that you can check that every model is well-formed.
3. Create some production system models that are representative for all the features in your language. The requirements for two valid models are specified below, and there should be a third invalid model to show that your constraints detect invalid models.
4. Write operational semantics (using EOL) that simulate the production system.
5. Write a report that includes a clear explanation of your complete solution and the modelling choices you made.

This assignment should be completed in groups of two if possible, otherwise individually is permissible.

Submit your assignment as a zip file (report in pdf, commented abstract syntax and operational syntax models, and simulator) on Blackboard before **Thursday, October 8th, 23:59h**. Contact Bentley Oakes (bentley.oakes@uantwerpen.be) if you have any issues.

2 Requirements

This section lists the requirements of the production system domain-specific language and the report. The language requirements are split into two sections: one on abstract syntax, and one on operational semantics. Make sure to test each requirement with test models!

2.1 Abstract Syntax

The *abstract syntax* of the DSL captures its *syntax* and *static semantics*. The requirements for the abstract syntax are:

1. A production system consists of the infrastructure with *conveyor belts* running between *machines*. *Workers* operate the machines while *items* are transferred on the belts. *Items* are processed and are either assigned by a quality check to be *accepted*, *rejected*, or *fixed*.

2. The belt network consists of a number of interconnected belt *segments*. The language must support the following segments:

   - **Straight** - A trivial belt segment which allows an item to move straight. Has one incoming and one outgoing segment.
   - **Split** - Allows an item to go straight, or to split off onto another belt. Has one incoming segment and two outgoing segments.
   - **Join** - Joins two segments. Has two incoming segments and one outgoing segment.
   - **Machine** - Similar to *Straights*, but can also be at the beginning or end of a belt. Depending on type has zero or one incoming segments and zero, one, or three outgoing segments, but is always connected to at least one segment.
     - Each *Machine* has a unique name, consisting of a single upper case letter, followed by zero or more lower case letters, ending with zero or more numbers.

3. Although this will not be allowed at run-time, the language should support more than one item to be present on a belt segment at a time.

4. There are two types of *Items* in this production system: *Cylinders* and *Cubes*. An *Item* must be on only exactly one segment.

5. There are a number of *Machines* which exist in this production system

   - *Arrival* - The *Arrival Machine* produces either *Cylinders* or *Cubes*. *Items* are produced when the *Machine* is operated.
   - *Assembly* - The *Assembly Machine* is actually two linked *Machines*. One for the *Cylinders* and one for the *Cubes*. These *Machines* are different segments, and are linked by a bi-directional association. An *Assembly Machine* handling *Cubes* must be linked to exactly one *Assembly Machine* handling *Cylinders* (and the reverse of course). The output of an *Assembly Machine* is an *AssembledItem*, which is itself an *Item*.
   - *Inspection* - The *Inspection Machine* inspects the *Item* (including *AssembledItem*), and determines whether the *Item* is to be accepted, fixed, or destroyed.

    - An *Inspection Machine* is still a type of *Segment*, but it must also have one output belt for the *Items* to fix, and one output belt for the *Items* to destroy.
  - *Receiver* - The *Receiver Machine* takes any incoming *Items* off the belt, for further processing.
  - *Fixer* - The *Fixer Machine* attempts to repair any defects in the *Item*.
  - *Incinerator* - The *Incinerator* destroys the *Item* on the belt.

6. Each of these *Machines* requires an *Operator* to operate. *Operators* have a *name*[1]. Each *Machine* can have at most one *Operator* be present, and the *Operator* must be present for the *Machine* to function.

7. These *Operators* also need a *schedule*, which will be defined in a second domain-specific language. This is so that each operator can have a different schedule in the production system. The requirements for this second language are:

   - A schedule is associated to an *Operator* by referring to the name of the *Operator*. Each *Operator* must have a schedule, and a schedule must have an *Operator*.
   - The schedule of an *Operator* tells them which *Machines* to operate, and for how many time steps. The *Operator* will start at the first *Machine* in the list, and operate them in order until the end of the list in which case the schedule will repeat. There must be at least one step where a machine is operated in each schedule.
   - Whenever the operator moves between two different machines (including when the schedule is repeated), there must be a step (of duration one) which represents the movement of the worker within the physical space. During this movement step, that operator will not operate any *Machine*.

2.2 Operational Semantics

In this part of the assignment, the semantics of the production system will be modelled, including the *Operators*, *Items*, and *Machines*. The goal is for the *Operators* to move between *Machines* and operate them, such that *Items* are assembled, inspected, fixed, received, or destroyed.

The specific requirements are:

- The simulation is broken up into a number of discrete steps. In each step, the *Machines* are operated if *Items* and *Operators* are present, the *Operators* are moved if needed, and then the *Items* are moved (concurrently).

[1] And hopes, dreams, fears, and rich social lives. But these qualities won't be modelled here, only their name.

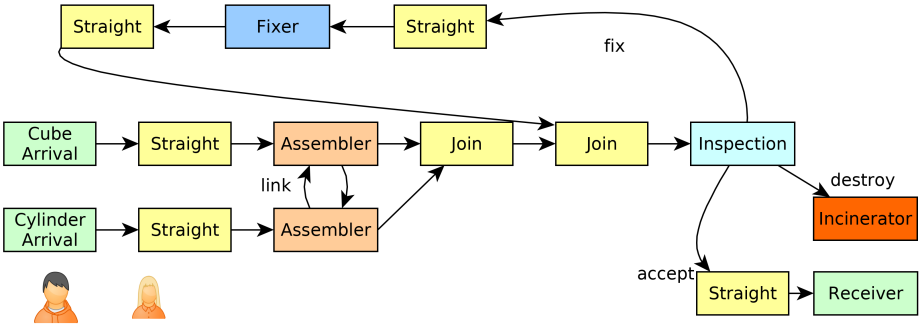

Figure 1: An example production system.

- In the initial step, all *Operators* are placed at their start *Machine*.
- If an *Operator* is scheduled to move to an occupied *Machine*, they must wait until the other *Operator* is finished.
- An *Item* is allowed to move to the next segment if no *Item* is present on that *Segment*, and the *Item* is not in an *Assembler* waiting for the linked *Machine* to obtain its *Item*.
- An *Arrival* cannot be operated if an *Item* is already on that *Segment*.
- In an *Assembler* when it operates, both *Items* are removed, and one is replaced with an *AssembledItem*.
- At a *Split*, *Cubes* will take the straight direction, while *Cylinders* will take the diverging direction.
- At a *Join*, one of the incoming *Items* is selected randomly to advance.
- At an *Inspector*, the chance of the *Item* being accepted (placed on the accepted belt) is 70%. The chance of requiring fixing is 20%. The chance of the *Item* requiring destruction is 10%.
- The simulation must produce a textual trace as in Figure 2.
- The simulation continues until some number of *AssembledItems* are accepted. Set this parameter such that the traces are long enough to show interesting behaviour.

3 Report

There are a number of requirements for the report. Above all, the marker must be able to read the report and have a clear understanding of all aspects of the assignment, without having to investigate the model files.

Specifically, the report must contain:

- A brief outline of how the abstract syntax, operational syntax, and scheduling models meet the requirements of the assignment, including any interesting decisions made.
- A brief description of the constraints present in your languages.
- Three example production systems.
  - Two valid, one invalid (doesn't meet the constraints).
- For each production system, show:
  - A small diagram (doesn't need to be elaborate, but enough to understand the trace)
  - The results of constraint checking on the invalid production system, and which constraint fails.
  - Interesting parts of the textual trace from the simulation, plus any extra explanation required to clearly understand the traces.

4 Useful Links and Tips

- metaDepth main page: `http://metadepth.org/`
  - `http://metadepth.org/papers/TOOLS.pdf`
  - `http://metadepth.org/Documentation.html`
  - `http://metadepth.org/Examples.html`
- Epsilion Object Language: `https://www.eclipse.org/epsilon/doc/eol/`
- Use an .mdc file to save time
  - See slide 47 of `https://metadepth.org/tutorial/tutorial.pdf`
- Can create two separate models, then import one into the other
  - See slide 94 of `https://metadepth.org/tutorial/tutorial.pdf`
- If assignments are failing with `Internal error: the value X is not a Y`, first assign the variable to `null` before performing the assignment. This is due to type checking.
- Use `if (x.isDefined())` to check for null
- Use `context ‘‘model_name‘‘` to change which model the EOL is executed in

Acknowledgements

Based on an earlier assignment by Simon Van Mierlo.

```
Start ProdSys
---=== ---=== Initial Setup === ---=== ---
Operator: 'Bob' arrives at machine: Cubearr
Operator: 'Alice' starts walking.
Operator: 'Ernie' arrives at machine: Inspection
Operator: 'Daniel' arrives at machine: Receiver
Operator: 'Charlie' arrives at machine: Cylassemb
---=== ---=== Step 1 === ---=== ---
Machine: 'Cubearr' produces a Cube
Operator: 'Daniel' still working for 1 more time steps at machine: Receiver
Operator: 'Bob' starts walking.
Operator: 'Alice' arrives at machine: Cubeassemb
Operator: 'Ernie' still working for 1 more time steps at machine: Inspection
Operator: 'Charlie' still working for 3 more time steps at machine: Cylassemb
Item: 'MD_b232aef93f724def9e2eb4b65e816a54' at: cube_arr
---=== ---=== Step 2 === ---=== ---
Operator: 'Daniel' starts walking.
Operator: 'Bob' arrives at machine: Cylarr
Operator: 'Alice' still working for 3 more time steps at machine: Cubeassemb
Operator: 'Ernie' starts walking.
Operator: 'Charlie' still working for 2 more time steps at machine: Cylassemb
Item: 'MD_b232aef93f724def9e2eb4b65e816a54' at: s1
---=== ---=== Step 3 === ---=== ---
Machine: 'Cylarr' produces a Cylinder
Operator: 'Daniel' arrives at machine: Incinerator
Operator: 'Bob' starts walking.
Operator: 'Alice' still working for 2 more time steps at machine: Cubeassemb
Operator: 'Ernie' arrives at machine: Inspection
Operator: 'Charlie' still working for 1 more time steps at machine: Cylassemb
Item: 'MD_b232aef93f724def9e2eb4b65e816a54' at: cube_assemb
Item: 'MD_90be4be6ec104663864325e9ffcf78c6' at: cyl_arr
etc. etc.
```

Figure 2: An example trace produced by my solution for the production system in Figure 1. Feel free to have more or less information than this.