Assignment 1
Production System Modelling in metaDepth

Randy Paredis
randy.paredis@uantwerpen.be

1 Practical Information

This assignment will make you familiar with the textual modelling tool **metaDepth**. You will learn to model two distinct domain-specific modelling languages (formalisms) that will be used for creating a simple factory.

The different parts of this assignment:

1. Implement the abstract syntax for your languages 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. Also mention possible difficulties you encountered during the assignment, and how you solved them. Don't forget to mention all team members and their student IDs!

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) on Blackboard before **19 October 2021, 23:59h**[1]. If you work in a group, only *one* person needs to submit the zip file, while all others *only* submit the report. Contact Randy Paredis if you experience any issues.

[1]Beware that BlackBoard's clock may differ slightly from yours.

2 Requirements

This section lists the requirements of the production system domain-specific languages. 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 (i.e., assembled) 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** - Identifies a bifurcation in the belt. 50% of the items will move to the first output segment and 50% will move to the second output segment. Hence, it has one incoming segment and two outgoing segments.
   - **Join** - Joins two segments. Has two incoming segments and one outgoing segment. The items will be outputted in order of arrival.
   - **Machine** - Similar to *Straights*, but potentially alter the given items. It 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. The *Join* segment is an exception for this rule, as it is allowed to have 2 items at the same time.

4. There are two types of (basic) *Items* in this production system: *Spheres* 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 *Spheres* or *Cubes*. *Items* are produced when the *Machine* is operated.

- **Assembly** - The *Assembly Machine* combines one *Cube* and one *Sphere* into one *AssembledItem* (which is itself an *Item*). Hence, it has two inputs and one output. The first input accepts *Cubes*, whereas the second input accepts *Spheres*.
  - It should be physically impossible to have a connection from an *Arrival* of a *Cube* to the *Assembly*'s *Sphere* input. Similarly, it should be impossible for *Spheres* to arrive at the *Assembly*'s *Cube* input. You may assume there are only conveyor belts between the *Arrivals* and the *Assembly*.
- **Inspection** - The *Inspection Machine* inspects the *Item* (including *AssembledItem*), and determines if an item must 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.
- **Loading Bay** - The *LoadingBay Machine* takes any incoming *Items* off the belt and stores them for future shipment.
- **Fixer** - The *Fixer Machine* attempts to repair any defects in the *Item*. For simplicity, you can ignore any internal workings for this machine.
- **Incinerator** - The *Incinerator* destroys the *Item* on the belt.

6. Each of these *Machines* requires an *Operator* to operate. *Operators* have a *name*[2], which should be unique. 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*.
   - 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*.
   - The schedule of an *Operator* tells them which *Machines* to operate, and for how many time steps. The *Operator* will start at the first

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

Machine in the list (which can optionally be `null`, if the *Operator* starts in a movement step), 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.

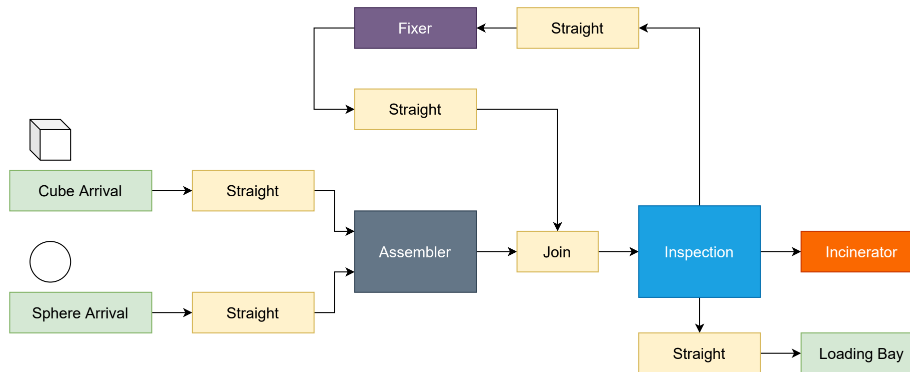

Figure 1: An example production system.

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 (if possible).
- 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 other shape to arrive. The *Join* is excempted for this rule, as it will allow at most two *Items* to be located on the segment.
- An *Arrival* cannot produce a new *Item* if an *Item* is already on that *Segment*.
- When the *Assembler* operates, both *Items* are removed, and replaced with an *AssembledItem*.

- At a *Split*, 50% of the *Items* will move to the first output and 50% to the second one. This can easily be ensured by having the *Split* consecutively toggle between both outputs. For instance, the first item will be outputted to the left, the second one to the right, the third one to the left again, etc...
- At a *Join*, the *Items* are ordered in a *first-in-first-out* order.
- At an *Inspector*, the correctness of an *Item* is determined as an integer in the range of $[0, 100]$. This value determines how the *Item* will be handled/outputted. 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, human-readable trace (as in Figure 2). You are free to have more or less information than given, as long as all required information for each step is outputted.
  - Clearly describe your trace file structures in your report!
- The simulation continues until some number of *AssembledItems* are accepted. Set this parameter such that the traces are long enough to show interesting behaviour.
- Think carefully about what information should be stored in the abstract syntax of the model, and which information is relevant to the simulator. You are free to choose how much information is stored in your model, as long as all information is useful in some way.

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. I.e., your model files will only be used as a support for your report, not the other way around!

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 all decisions and assumptions 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:

```
===== Start Production System =====
Operator: 'Bob' arrives at machine: Cubearr
Operator: 'Alice' starts walking.
Operator: 'Ernie' arrives at machine: Inspection
Operator: 'Daniel' arrives at machine: Receiver

===== STEP 0 =====
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: Asm
Operator: 'Ernie' still working for 1 more time steps at machine: Inspection
Item: 'MD_ea58cb3930414346b17eb8531c17481d' at: cube_arr

===== STEP 1 =====
Operator: 'Daniel' starts walking.
Operator: 'Bob' arrives at machine: Sphrarr
Operator: 'Alice' still working for 3 more time steps at machine: Asm
Operator: 'Ernie' starts walking.
Item: 'MD_ea58cb3930414346b17eb8531c17481d' at: s1

===== STEP 2 =====
Machine: 'Sphrarr' produces a Sphere
Operator: 'Daniel' arrives at machine: Incinerator
Operator: 'Bob' starts walking.
Operator: 'Alice' still working for 2 more time steps at machine: Asm
Operator: 'Ernie' arrives at machine: Inspection
Item: 'MD_ea58cb3930414346b17eb8531c17481d' at: assemb
Item: 'MD_ce324ac4bb714294a3897070c3ec231d' at: sphere_arr

===== STEP 3 =====
Operator: 'Daniel' still working for 1 more time steps at machine: Incinerator
Operator: 'Bob' arrives at machine: Cubearr
Operator: 'Alice' still working for 1 more time steps at machine: Asm
Operator: 'Ernie' still working for 1 more time steps at machine: Inspection
Item: 'MD_ea58cb3930414346b17eb8531c17481d' at: assemb
Item: 'MD_ce324ac4bb714294a3897070c3ec231d' at: s2

...
```

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

- A small, graphical diagram (doesn't need to be elaborate, but enough to understand the trace). This can be as simple or as fancy as you want. PlantUML (https://plantuml.com/), GraphViz (https://graphviz.org/) and DrawIO (https://draw.io/) are excellent tools to create such a diagram.
- The results of constraint checking on the invalid production system, which constraint(s) fail(s) and why.
- 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/
- A package for the Atom text editor (https://atom.io/), that allows a very basic syntax highlighting for both `EOL` and `metaDepth` is available: http://msdl.uantwerpen.be/people/hv/teaching/MSBDesign/assignments/metadepth.zip
  Any updates and bugfixes made to this package are allowed, and free to be mentioned in your submission/report or via email.
- 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
- While you can do a lot with metaDepth, its documentation is limited in the available features. The TOOLS paper and the tutorial provide some insights, but a lot are hidden in the source code. Below is a short summary of some possibly useful features:
  - `Node`s can be marked `abstract` to prevent users from instantiating them.
  - Attributes can be marked as an *identifier* (using `{id}`) to ensure global uniqueness. This is similar to a database's ID. An example: `name: String{id};`
  - Collection attributes can be marked `unique` to prevent duplicate items and `ordered` to keep the order of the elements.

```
items: Item[*] {unique, ordered};
```

  - Builtin attribute types are: `int`, `double`, `boolean`, `String` and `Date`. Any collection of these attributes is also possible, as well as custom types.
  - Enumerations can be created using:

    ```
    enum MyEnum {VALUE1, VALUE2, VALUE3};
    ```

    They must be in the `Model`-scope and can be compared in EOL as simple strings on their values.

- 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 Bentley Oakes.