Assignment 2
Modelling in AToMPM

Randy Paredis
randy.paredis@uantwerpen.be

1 Practical Information

This assignment will make you familiar with the visual modelling tool **AToMPM**. You will learn to create meta-models and abstract and concrete syntaxes for a domain-specific modelling language (formalism) concerning production systems (factories).

The different parts of this assignment:

1. Implement the abstract syntax of your language in AToMPM.
   - The formalism used will be `/Formalisms/LanguageSyntax/SimpleClassDiagram`
   - You can also quickly create new formalisms with the *create new formalism* button.
2. Enrich the abstract syntax with constraints so that you can check that every model is well-formed.
3. Create a concrete syntax, and generate a modelling environment by compiling the metamodel and the concrete syntax model. Do this incrementally.
   - The formalism for this part will be `/Formalisms/LanguageSyntax/ConcreteSyntax`
4. 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.
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 + abstract and concrete syntax models) on Blackboard before **9 November 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.

2 Requirements

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

2.1 Abstract Syntax

There are no modifications in this section from Assignment 1.

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.

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

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:

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

- 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 *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.

2.2 Concrete Syntax

Notations in production systems modelling are not standardized. Therefore you will have a lot of freedom coming up with your own notation. The only requirements are that:

- Your notation does not need to be beautiful, but it must be clear and understandable.
- The license for downloaded images must be respected. For example, `flaticon.com` requires textual attribution which can be placed in your report.

Figure 1 shows an example production system from the first assignment, and one representation of it. Note that your solution should be somewhat similar.

As well, the actions, mappers, and parsers of AToMPM must be used to improve the user experience of modelling the production system:

- Display the percentage chance of acceptance, rework, and failure on inspection machines, controlled with attributes on the machine instance.
- Display other useful information as you see fit (such as the name of machines and operators).
- Model an action that automatically "snaps" a segment when it is connected to another segment.
  - As in, when two segments are connected, one moves directly adjacent to the other.
  - `Formalisms/Pacman` has an example of this in the `Positionable` class.

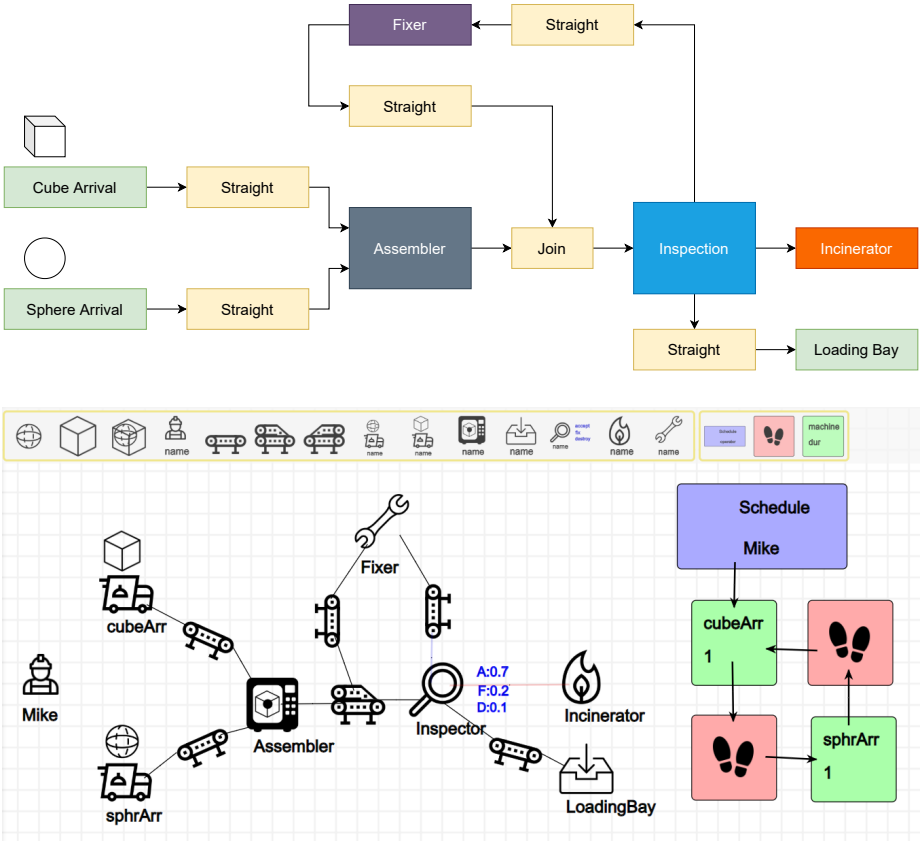

Figure 1: An example production system and its model in AToMPM.

3 For the Next Assignments

The next assignments will all utilize AToMPM for various model transformations. Therefore:

- Spend time becoming familiar with AToMPM concepts and interface.
- Report issues, bugs, annoyances, and suggestions to Randy Paredis[3].
- Think carefully about your solution, and spend extra time improving the concrete syntax. To prevent issues in future assignments, make sure you keep your abstract syntax as close as possible to your solution for the first assignment
- Look at the AToMPM documentation for how to use transformations, and if possible begin experimenting.
  - The next assignment will use transformations to implement the operational semantics of the production system.

4 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, concrete syntax, and example models meet the requirements of the assignment
  - This may include metamodels, diagrams, (pseudo-)code, etc. as needed to provide the essential details of the assignment.
- A discussion of any interesting decisions and assumptions made.
- A discussion of possible improvements to the abstract/concrete syntax.
- A brief description of the constraints present in your languages.
- Three example production systems.
  - Two valid, one invalid (which doesn't meet the constraints).
- For each production system, show:
  - A figure of the production system within AToMPM.

[3] As AToMPM is nearing its end of life, these will not be solved, but rather marked as checks for the next visual meta-modelling tool.

- The results of constraint checking on the invalid production system, and which constraint fails.
- These production systems should be *in medias res* (in the middle of execution). This means that there should be items on belts and at machines, and operators at machines. This shows the validity of your concrete syntax.

5 Useful Links and Tips

- AToMPM main page: https://atompm.github.io/
- Download and code: https://github.com/AToMPM/atompm
- Documentation: https://atompm.readthedocs.io/en/latest/

Acknowledgements

Based on an earlier assignment by Bentley Oakes.

Icon authors from www.flaticon.com:

- Sphere - https://www.flaticon.com/authors/good-ware
- Cube, Receiver - https://www.flaticon.com/authors/smashicons
- Belts, Machine, Inspector, Incinerator - https://www.flaticon.com/authors/freepik
- Assembler - https://www.flaticon.com/authors/catalin-fertu
- Fixer - https://www.flaticon.com/authors/srip
- Walk - https://www.flaticon.com/authors/vitaly-gorbachev