Assignment 2
Concrete and Abstract Syntax in AToMPM

Joeri Exelmans
joeri.exelmans@uantwerpen.be

1 Practical Information

This assignment will make you familiar with the visual modelling tool **AToMPM** ("A Tool for Multi-Paradigm Modeling"). You will create a meta-model for the abstract syntax of your language (similar to the previous assignment, but this time using a *visual language*), and define additional constraints (similar to the previous assignment). Further, you will define a visual concrete syntax for your language. You will not yet define an operational semantics; you will do this in the next assignment. Tabel 1 compares MetaDepth and AToMPM.

	MetaDepth	**AToMPM**
concrete syntax	textual, auto-generated	visual (topological), explicitly specified
abstract syntax	class diagram-like supports multi-level modeling	class diagrams
constraint language	EOL	JavaScript
operational sematics	EOL (imperative)	PyTCore (model transformation)

Table 1: Comparison between MetaDepth and AToMPM

The different parts of this assignment:

1. Create a new formalism in AToMPM with the *create new formalism* button. This will create two empty models in the following files:
   - `/Formalisms/NAME/NAME.model`, the meta-model for your abstract syntax.
   - `/Formalisms/NAME/NAME.defaultIcons.model`, the meta-model for your concrete syntax.
2. Implement the abstract syntax of your language in AToMPM by editing the `.model` file.
3. Enrich the abstract syntax with constraints.

4. Specify a concrete syntax, by editing the `.defaultIcons.model` file, and generate a modelling environment by compiling the metamodel and the concrete syntax model.

5. Create some example traffic network models that are representative for all the features in your language. We expect one valid models, and two invalid models.

6. Write a report.

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 **7 November 2023, 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 Joeri Exelmans if you experience any issues.

2 Requirements

This section lists the requirements of the traffic DSL. The language requirements are split into two sections: one on abstract syntax, and one on concrete syntax.

2.1 Abstract Syntax

The requirements for Abstract Syntax and Constraints remain identical to those of the previous assignment, but we have added some hints in yellow boxes, like this one.

The *abstract syntax* of the DSL can be thought of as a set of constraints that instances of your language must conform to. These include multiplicities on types, relations between types, and additional constraints (to compensate for lack of expressiveness of multiplicities).

Our road network language consists of the following elements:

- **RoadSegment** - A uni-directional piece of road. It can contain at most 1 car. A Segment has one output (that connects to the next segment), and one input (that connects to the previous segment).
- **Split** - Like a RoadSegment, but with **two outputs**. A car can move via either output to one of the two connecting segments.
- **Join** - Like a RoadSegment, but with **two inputs**. A car can move from either input .
- **Generator** - A source for new cars. It only has one output, that connects to a RoadSegment. A generator itself cannot contain cars. It models input to the System under Study from the Environment.

[1] Blackboard's clock

- **Collector** - Where cars disappear. It only has one input, that connects to a RoadSegment. A collector itself cannot contain cars. It models output from the System under Study from the Environment.
- **Connection** - A connection from an output (of a Generator, RoadSegment, Join or Split), to an input (of a Collector, RoadSegment, Split, or Join).
- **Car** - A car.

> **Hint:** Whether a Car is a class, or just an attribute of an element that can hold a car, is up to you. However, it is easier to define a concrete syntax in the latter case (see later).

- **Schedule** - Defines a sequence of steps to be made, encoded as an ordered sequence of Connections.

> **Hint:** It's not possible to create (ordered) lists of instances in AToMPM. Therefore, we recommend giving every Connection a 'name' attribute, and the schedule then contains a list of Connection names.

> **Hint:** Both classes and associations can be given attributes in AToMPM.

> **Hint:** Introducing additional abstract type(s) may make your solution cleaner. Further, AToMPM supports multiple inheritance **but cannot resolve the *diamond problem*** [a] **!**
>
> [a] https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

2.1.1 Constraints

> Also here, no modifications since Assignment 1, except that this time, you will have to write constraints in JS.

Further, you should model the following constraints:

- A segment's output cannot connect directly to its own input. Cycles are allowed however, but must consist of at least two elements.
- There must be at least one Generator and at least one Collector.
- There must always be exactly one Schedule.
- Every segment must be reachable from a Generator.
- From every segment, a Collector must be reachable.

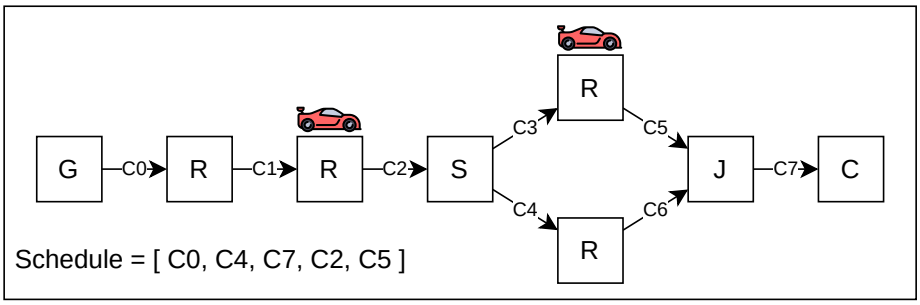


Figure 1: A possible visual concrete syntax.

2.2 Concrete Syntax

In this assignment, you have a lot of freedom to come up with your own notation. But we do require the following:

- Most importantly, the notation should be clear and understandable. Use intuitive icons, colors, etc.
- All attributes of all elements must be visible without having to open the properties dialog.

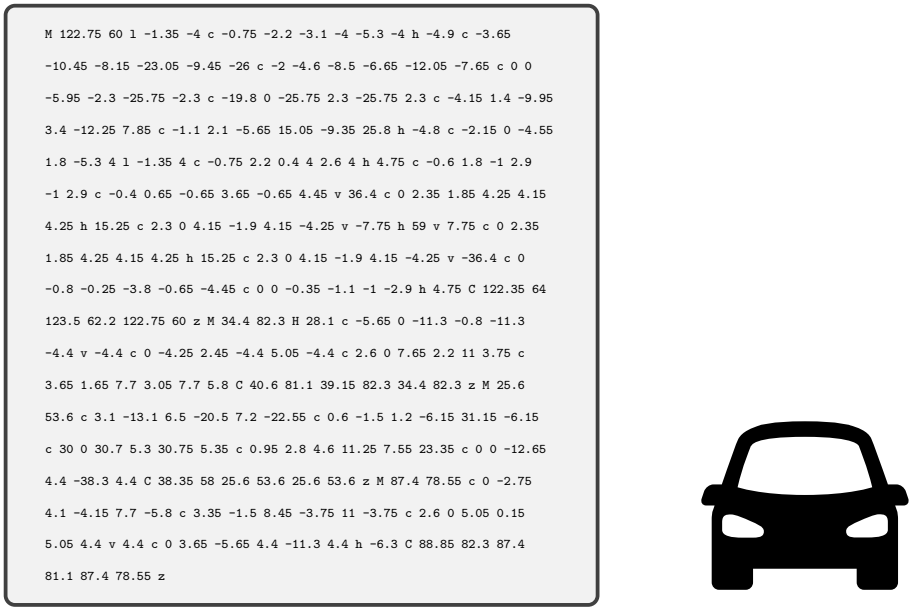
  Figure 1 shows a possible visual concrete syntax (taken from 1st assignment): the complete model can be understood from this image. You can try to base yourself on this syntax, and enrich it further with color, etc.
- If a traffic network element has a Car on it, then an icon of a Car must be visible "on" the icon of the traffic network element (instead of having an arrow from a Car to a network element).

  This can be implemented in the following ways:

  - (easier) If you decide that a Car is just an attribute of a network element, you can show/hide a shape or path by setting `fill-opacity` to 1 or 0 in your mapper(s). This does not work for Images, but you can use (SVG) 'Paths' in AToMPM. For instance, Figure 2 shows an SVG path and the (car) shape it draws. You can use (copy) this path in AToMPM (the "segments" attribute).

    Many SVG icons define a single path in their source file. For instance, the one in Figure 2 was extracted from http://cdn.onlinewebfonts.com/svg/img_374502.svg.

    If you feel creative, you can create a path yourself using https://yqnn.github.io/svg-path-editor/.
  - (harder) If you decide that every Car is a separate instance (of a class), then you must update the position of the Car after connecting it to a network element. The 'Positionable' class of the 'Pacman' example, included in AToMPM, demonstrates how this can be done:



(a) (b)

Figure 2: An SVG path (a) and the shape drawn by it (b).

1. Add a 'position' attribute to the abstract syntax
2. Add an action, triggered by the 'post-connect' event, to the abstract syntax, that sets the position to the position of your element
3. Define a mapper and parser in the concrete syntax that synchronizes the position attributes in concrete and abstract syntax.

3 Report

You will write a report containing:

- An explanation of your workflow, and motivations for decisions made.
- An overview of your solution:
  - A screenshot of your abstract syntax meta-model + for every class and association, the attributes you added.
  - It is not necessary to add a screenshot of your concrete syntax definition (the example models you will provide, will give me a much better impression).
  - The full code of the constraints written.

- Three example models of your choice (you are allowed to re-use some models from assignment 1).
  - **One** conforming, **two** non-conforming.
  - For each model, show a screenshot of AToMPM with this model.
  - For each of the non-conforming models, a different constraint needs to fail (in AToMPM, as soon as one constraint fails, the constraint checking ends). Show me which constraint fails, and explain why.

4 Grading

We reserve the right deviate, but roughly, you can expect the following:

- (25 %) abstract syntax meta-model
- (25 %) extra constraints
- (25 %) concrete syntax definition
- (25 %) report

5 Useful Links and Tips

- Use AToMPM version 0.10.0rc3. (Version 0.9.0 is fine but doesn't work with Python >= 3.10)
  - Docker container: https://github.com/AToMPM/atompm/pkgs/container/atompm
  - Linux/Mac users can download the source code: https://github.com/AToMPM/atompm/archive/refs/tags/v0.10.0rc3.tar.gz
    - Install NodeJS and NPM
    - Then run `npm i` in the source directory
    - Then, run `node httpwsd.js`
    - Navigate to http://localhost:8124/atompm
    - (We don't need the Python model transformation backend yet, so it is not yet necessary to install Python and the Python dependencies.)
  - Windows users can try the "portable" version, which includes all dependencies: https://github.com/AToMPM/atompm/releases/download/v0.10.0rc3/atompm-portable.zip
- FSA Example from in-class tutorial: http://msdl.uantwerpen.be/people/hv/teaching/MSBDesign/examples/AToMPMExample2324-myFSA.zip
  - Put the directory 'myFSA' in `atompm/users/<you>/Formalisms/`.

- AToMPM tutorial (covers everything): https://atompm.readthedocs.io/en/latest/new_language.html
  - Supported datatypes (for attributes): https://atompm.readthedocs.io/en/latest/new_language.html#attributes
    - e.g., to create a list of integers, use `list<int>`
    - One type of bug that can be hard to track down is when you set the default value of an integer attribute to `"0"` (which is a JSON string literal), but it should be `0` (i.e., JSON number literal). Same for booleans.
  - The section explaining the API for writing constraints is especially useful: https://atompm.readthedocs.io/en/latest/new_language.html#action-library
    - The `getNeighbors()` API call gives you the IDs of the incoming or outgoing *links* when called on a node ID. Call `getNeighors()` a second time (with the same 'dir' parameter) on the link ID, to get the node that the link connects to. See the `GC_EveryStateReachable` constraint in the myFSA example.
    - Debug the JavaScript code of your constraints by adding `console.log()` statements. Output will be printed in the terminal that started the `node httpswd.js` process.
- AToMPM has a "funky" UI:
  - To create a new instance of a class, use **right-click**.
  - To edit the properties of an instance, use **middle-click** on the object while it IS NOT selected (Mac: **Shift+left-click**), or the **insert-key** on your keyboard (while the object IS selected).
- Don't forget to **compile** your abstract and concrete syntax after changing them. Also, when changing your abstract or concrete syntax, you have to re-create your instances from scratch...
- Expect some crashes, so save often!
  - If AToMPM crashes, make sure to kill all the `node` processes before starting it again!

Acknowledgements

Based on an earlier assignment by Randy Paredis.