Creating a traffic DSL in MetaDepth

Joeri Exelmans
joeri.exelmans@uantwerpen.be

1 Practical Information

In this course, you will create your own Domain-Specific Languages (DSLs) to simulate and analyze (car) traffic. The goal of this first assignment is to create a meta-model for your DSL, to create instances of this meta-model and to verify conformance between (instance) model and meta-model. You will also specify the operational semantics of this language. You will use the textual modeling tool MetaDepth, and its action + constraint language EOL (Epsilon Object Language).

The different parts of this assignment:

1. Implement the abstract syntax for your language(s) 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 instance models that are representative for all the features in your language. The requirements for two conforming models are specified below, and there should be a third non-conforming model to show that your constraints detect non-conforming models.
4. Write operational semantics (using EOL) that, given a conforming model, simulates one "step" (in the operational/simulation semantics), producing a new conforming model.
5. Write a report that includes a clear explanation of your complete solution and the modeling 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. Working individually is also allowed.

Submit your assignment as a zip file (report in pdf + MetaDepth files for abstract syntax and operational semantics) via Blackboard before **24 October 2023, 23:59h**[1]. If you work in a group, only *one* person needs to submit the zip file, while the other person *only* submits a text file containing the name of the partner. Contact Joeri Exelmans if you experience any issues.

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

2 Requirements

You will develop a domain-specific modeling language for road networks and car traffic. Your language will capture topological information ("what is connected to what?") about roads, and the cars that navigate on them.

You will first develop the abstract syntax of your language, followed by the operational semantics.

2.1 Abstract Syntax

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).

The abstract syntax of our road network language consists of the following types:

- **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.
- **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.
- **Schedule** - Defines a sequence of steps to be made, encoded as an ordered sequence of Connections. The behavior encoded in a step will be explained in the section on Operational Semantics.

Hint: Introducing additional abstract type(s) may make your solution cleaner. MetaDepth supports multiple inheritance!

2.1.1 Constraints

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.

2.2 Operational Semantics

The operational semantics consist of a *step* function, which takes one valid model and transforms it to a next valid model. Roughly speaking, during a step, one car may move along a Connection from one elements to another. A move is only possible if: (1) the previous element has a Car available, and (2) the next element has enough remaining capacity. *Availability* is defined as follows:

- **Generator** Always has a car available.
- **Collector** Never has a car available.
- **RoadSegment, Split, Join** have a car available if there is currently a car on it.

Capacity is defined as follows:

- **Generator** Never has capacity (capacity 0).
- **Collector** Always has capacity (infinite capacity).
- **RoadSegment, Split, Join** have capacity 1. Only if there is currently no car on it, there is remaining capacity.

The schedule defines along which Connections how cars will be moved. The step function always pops the *last* Connection from the Schedule, and attempts to move a car along it. If the Schedule is empty, a step has no effect. Hence, in this assignment, the operational semantics are *deterministic*.

Note: MetaDepth seems to give errors when removing an element from an ordered collection. You can use this workaround: leave the Schedule unchanged, and instead keep the index of the next Connection to be stepped (like in the FSA example).

In the implementation of the operational semantics in EOL, please include some print statements to trace the behavior.

Note: MetaDepth does not automatically verify if your model still conforms to your meta-model after executing a step. You should manually instruct MetaDepth to perform this check after every step.

Figure 1 and Figure 2 show example models and their execution traces in a concrete visual 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 Class Diagram of your abstract syntax meta-model.
    Feel free to use DrawIO (https://draw.io/), PlantUML (https://plantuml.com/), MS Paint, or a scan of a sketch on paper (as long as it is readable!)
  - The full code of the constraints written.
- Three example models of your choice:
  - Two conforming, one non-conforming (i.e., violates at least one of the constraints).
  - For each model, show:
    - A small, graphical diagram in the style of Figures 1 and 2.
    - For the conforming models, the textual output of a number of execution steps in MetaDepth.
    - For the non-conforming model, the results of constraint checking. Explain which constraint(s) fail(s) and why.

4 Grading

- (25 %) abstract syntax meta-model
- (25 %) extra constraints
- (25 %) operational semantics
- (25 %) report

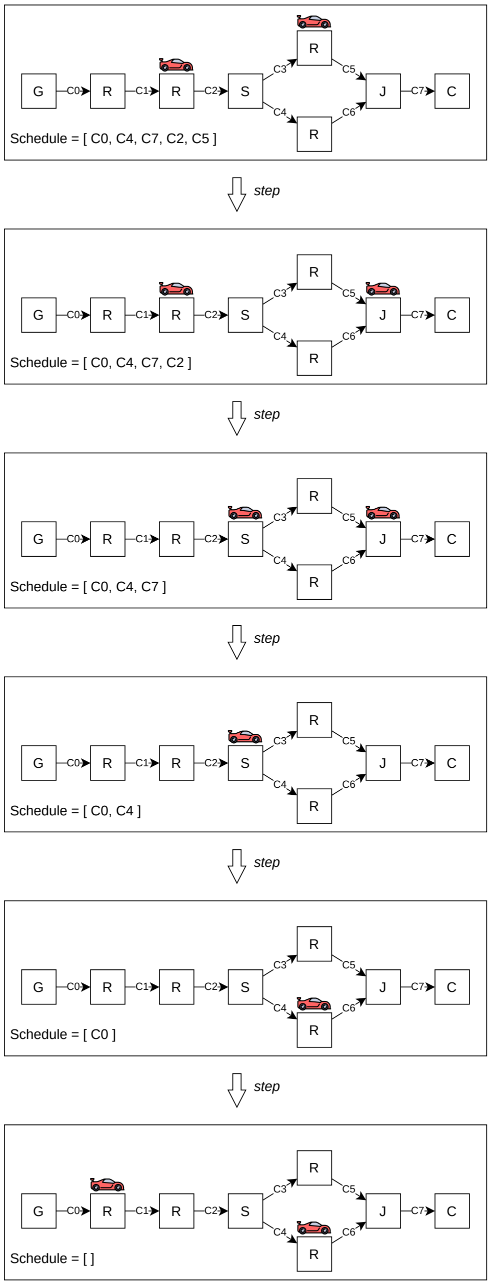

Figure 1: An example model and its execution trace. G: Generator, R: Road-Segment, S: Split, J: Join, C: Collector, C[0-7]: Connection

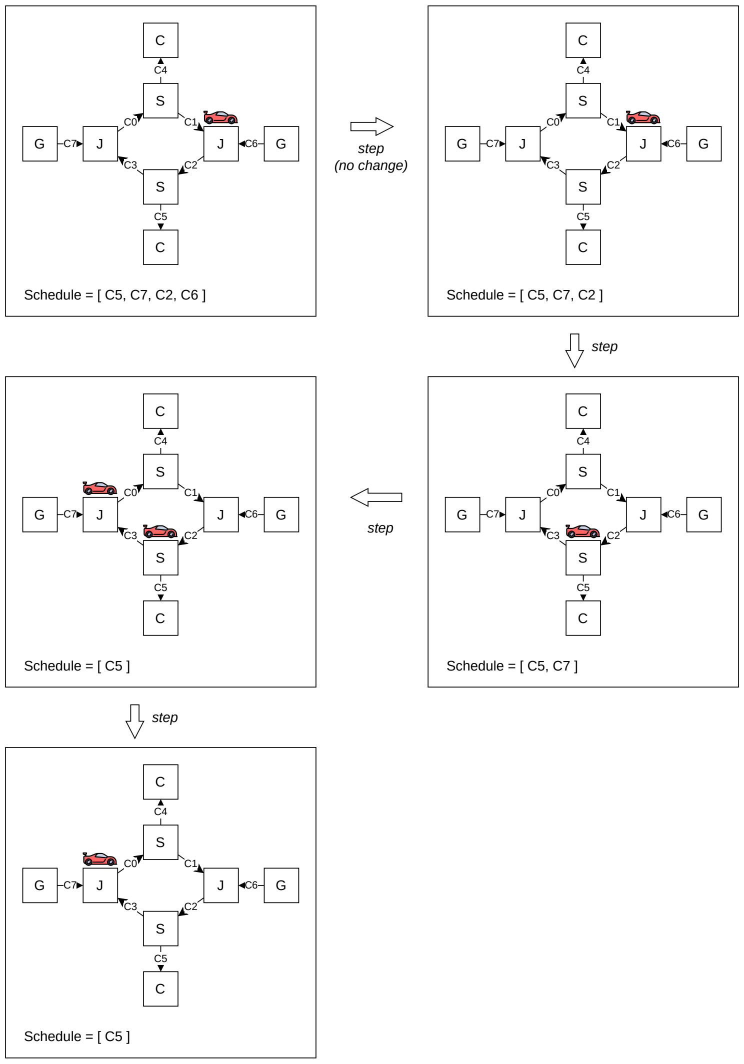

Figure 2: Another example.

```
Model FSA {
  Node State {
    outs: State[*]
    ins: State[*];
    // ...
  }
  Edge Transition(State.outs, State.ins) {
    //              ^ source    ^ target
    // ...
  }
}

FSA example {
  State A{}
  State B{}
  Transition T(A,B) {}
}
```

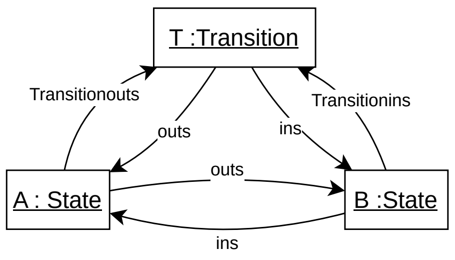

Figure 3: Using Edges in MetaDepth

5 Useful Links and Tips

- Main resources for this assignment:
  - These examples: http://msdl.uantwerpen.be/people/hv/teaching/MSBDesign/examples/mdepthExamples202324.zip

    The examples are:
    - **fsa** - A simple finite state automaton. Demonstrates abstract syntax, constraints, and operational semantics.
    - **fsa-edges** - Same as 'fsa', but uses 'Edge' for Transitions, leading to much more compact code. ~~TIP: Use 'Edge' for Connections!~~ Figure 3 shows an example of using Edges, and the relations that are created.

    Note: (21 October) MetaDepth appears to not work correctly when combining Edges and inheritance. Therefore, I recommend you to use Nodes (instead of Edges) for Connections.

    - **abstract** - Demonstrates usage of abstract types, and various EOL features.

    The examples demonstrate *all* MetaDepth and EOL features necessary to complete this assignment!
  - EOL documentation: https://www.eclipse.org/epsilon/doc/eol/
- Extra tips:
  - VS Code syntax highlighting package for EOL: http://msdl.uantwerpen.be/people/hv/teaching/MSBDesign/mdepth-0.0.1.vsix
  - **Very useful for this assignment:** In your EOL file, you can define methods on types as follows:

```
operation Generator hasCarAvailable(): Boolean {
  return true;
}
operation RoadSegment hasCarAvailable(): Boolean {
  return (self.car.isDefined());
}
```

  You can define the same method on different types. MetaDepth support polymorphism through dynamic binding, meaning that it will look up the right method to call at run-time.

  - `Nodes` 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};
    ```

    See the EOL documentation for the precise meaning of these annotations.
  - Builtin attribute types are: `int`, `double`, `boolean`, `String` and `Date`. Any collection of these attributes is also possible, as well as custom types.
  - 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.
  - In your EOL code, any value/type/object can be printed. This is useful for debugging.
  - Use `if (x.isDefined())` to check for `null`.
  - Using the name "in" for attributes/types/instances works in MetaDepth, but it is a reserved keyword in EOL, causing the parsing of your EOL expressions to fail, so you cannot use this name.
  - Use `context "model_name"` to change which model the EOL is executed in.
  - Use an .mdc file to save time
    * See slide 47 of https://MetaDepth.org/tutorial/tutorial.pdf

- Additional MetaDepth resources:

- Tutorial I gave in class (based on the FSA example): `http://msdl.uantwerpen.be/people/hv/teaching/MSBDesign/tutorial202324.zip`
- Main page: `http://MetaDepth.org/`
- Official examples: `http://MetaDepth.org/Examples.html`
- Official documentation: `http://MetaDepth.org/Documentation.html`
- TOOLS paper: `http://MetaDepth.org/papers/TOOLS.pdf`
- 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.

  Alternatively, you can use JavaScript syntax highlighting for EOL, and it will look OK-ish.

Acknowledgements

Based on an earlier assignments by Randy Paredis, Simon Van Mierlo, Bentley Oakes and Claudio Gomes.