Assignment 1
Waterway Modeling in metaDepth

Joeri Exelmans
joeri.exelmans@uantwerpen.be

1 Practical Information

The goal of this assignment is to design a meta-model for your a domain-specific (modeling) language (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.

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", 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 + commented abstract syntax and operational syntax models) on Blackboard before **19 October 2022, 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 a network of waterway traffic. Your language will capture topological information ("what is connected to what?") about waterways, and the watercraft that navigates on it.

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.

2.1.1 Waterway network abstract syntax

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

Watercraft - A manmade "thing" (e.g., boat, ship, drone, ...) that at any time must be located on exactly one Segment or Confluence.

Segment - A "piece" of waterway. A Segment instance can contain at most 1 Watercraft instance. A Segment has one "out"-connection (that connects to another Segment), and one "in"-connection (that also connects to another Segment).

Source - A special segment type that only has an "out". It is a place where new Watercraft is spontaneously generated. A Source has an "out" connection to a Segment, onto which it puts generated Watercraft, as we'll see in the operational semantics. A Source keeps a counter of how many Watercraft instances it has generated so far. A Source has a *rate*, which is a strictly positive integer parameter, indicating the number of steps to wait before another Watercraft is generated.

Sink - A special segment type that only has an "in". It is a place where Watercraft is destroyed/consumed. A Sink keeps a counter of how many Watercraft instances it has consumed so far.

Note: Sources and Sinks cannot contain any Watercraft.

Confluence - A special segment type with two in-connections (in_0 and in_1) and two out-connections (out_0 and out_1). Just like an ordinary segment, a confluence can contain (at most) 1 Watercraft instance. A Confluence has a *mode*, which is either 0 or 1.

We'll see that, either traffic can flow from in_0 to out_0, or from in_1 to out_1, both not both simultaneously. (Also, traffic can never flow from in_0 to out_1 or from in_1 to out_0.)

For all of these connections, an "in" must connect to an "out" (and vice-versa): If an element A's "out" connects to another element B, then B's "in" must connect to A. Further, a segment's output is not allowed to be connected to the same segment's input.

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

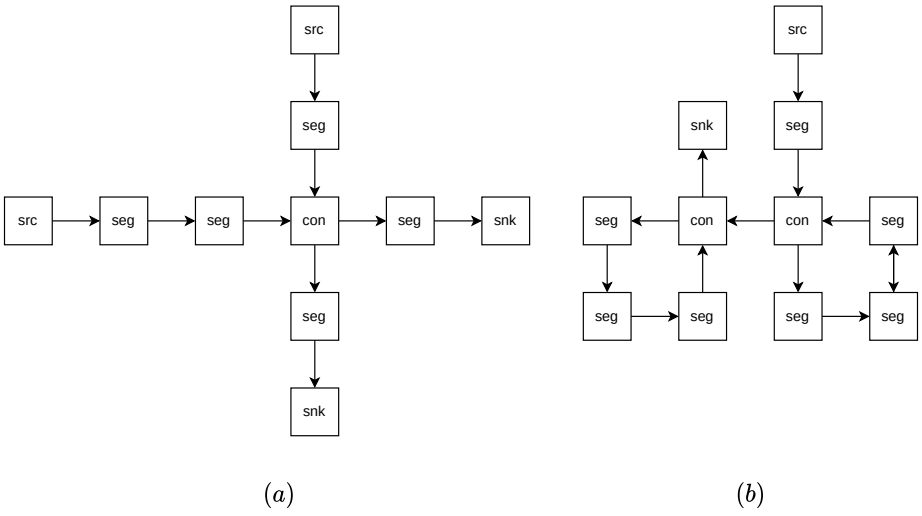

Figure 1: Some example waterway networks.

2.2 Operational Semantics

The semantics of our waterway network-language can be understood as a sequence of discrete *steps*. Every step has as input a valid instance, and during the step, this this instance is modified (in-place) to produce a new valid instance at the end of the step. In this assignment, we will specify the logic of step execution in an operational (think: procedural) manner, using the EOL language.

The implementation of the operational semantics of a step should be as follows:

- Every segment (i.e., ordinary Segment, Sink, Source, Confluence) is allowed to perform one micro-step.
- A segment cannot perform a micro-step until all of its outputs have performed a micro-step first. As a consequence, the first segment that will perform a micro-step in your model will always be a Sink.

  Hint: In EOL, you can use a Map to keep track of the segments that have already micro-stepped during the current step.

- Depending on the type of segment, a micro-step looks as follows:

  Sink . If the Sink's input segment has a Watercraft *available* (what this availability means, is explained later), it consumes the Watercraft, removing it from the input segment, and incrementing the Sink-counter by 1.

  (ordinary) Segment . If the Segment's input has a Watercraft available, and this Segment has no Watercraft, then this Watercraft is moved onto this Segment.

  Confluence .

  - If the Confluence already has a Watercraft on it, nothing happens (not enough space).
  - Otherwise: If the current mode of the Confluence is x, that means that most recently (in a previous step), a Watercraft from input i_x was allowed passage. Because a Confluence implements fair scheduling, in this step, it will give priority to input $1-x$ (in other words, the input that was **not** most recently allowed passage). If input $1-x$ has a Watercraft on it, the Watercraft is moved from that input onto the Confluence, and the mode is updated to $1-x$. If input $1-x$ does not have a Watercraft on it, but input x does have a Watercraft on it, then that Watercraft is moved from that input onto the confluence (and the mode remains equal to x).

  Source . Does nothing.

- Depending on the type of segment A, it has a Watercraft available for another segment B if:

  Source . True it has been $n \leq r$ steps since a Watercraft was produced by the Source, where r is the Source's *rate*. A Source also has a counter that counts the number of watercraft that has been produced.

  (ordinary) Segment . True iff the segment A has a Watercraft on it.

  Confluence . True iff the Confluence A has a Watercraft on it, and B is output out_x of A, where x is A's current mode.

  Sink . Never.

- Recap (this is not new information): A Confluence's *mode* serves two purposes within the same step: First, when its outputs are being micro-stepped, the mode determines the output Watercraft should use to leave the Confluence, Next, (still in the same step) when the Confluence is being micro-stepped, it determines which input has the *lowest* priority.

- For every step, a textual, human-readable trace of the actions executed during the step must be printed (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 output.

– Clearly describe your trace file structures in your report!

Figure 3 shows the execution of an example model.

Think carefully about what information should be stored in the abstract syntax, and what information is only temporarily relevant during the execution of a step.

```
===== STEP =====
Moved from 'seg0b' to 'snk0' (destroyed)
Moved from 'seg0a' to 'con'
Moved from 'src0' to 'seg0a' (created)
===== STEP =====
Moved from 'con' to 'seg0b'
Moved from 'seg1a' to 'con'
===== STEP =====
Moved from 'seg0b' to 'snk0' (destroyed)
Moved from 'con' to 'seg1b'
Moved from 'seg0a' to 'con'
Moved from 'src0' to 'seg0a' (created)
Moved from 'src1' to 'seg1a' (created)
```

Figure 2: An example human-readable trace produced by my solution for the production system in Figure 3. Feel free to have more or less information than this.

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.

3 Report

There are a number of requirements for the report. Above all, it must convey 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 and operational syntax, including all decisions and assumptions (if any) made.
- A brief description of the constraints present in your languages.
- Three example models:
  - Two conforming, one non-conforming (i.e., violates at least one of the constraints).

- For each model, show:
  - 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. DrawIO (https://draw.io/), PlantUML (https://plantuml.com/), GraphViz (https://graphviz.org/) and MS Paint (https://98.js.org/) are excellent tools to create such a diagram.
  - The results of constraint checking on the non-conforming model, 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
- The Epsilon Object Language (EOL) is fairly rich: 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.

  Alternatively, you can use JavaScript syntax highlighting for EOL, and it will look kindof OK.
- Use an .mdc file to save time
  - See slide 47 of https://metadepth.org/tutorial/tutorial.pdf
- While you can do a lot with metaDepth, its documentation is fairly limited. 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.

- In your EOL file, you can define methods on types as follows:

  ```
  operation Source hasWatercraft(): Boolean {
    // warning: read the requirements
    // before copying this :)
    return true;
  }
  operation Segment hasWatercraft(): Boolean {
    return (self.watercraft.isDefined());
  }
  ```

  You can define the same method on different types, and call it in a polymorphic fashion. Only at run-time is it checked whether an object actually has the method you're trying to call.

- 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 assignments by Randy Paredis, Simon Van Mierlo, Bentley Oakes and Claudio Gomes.

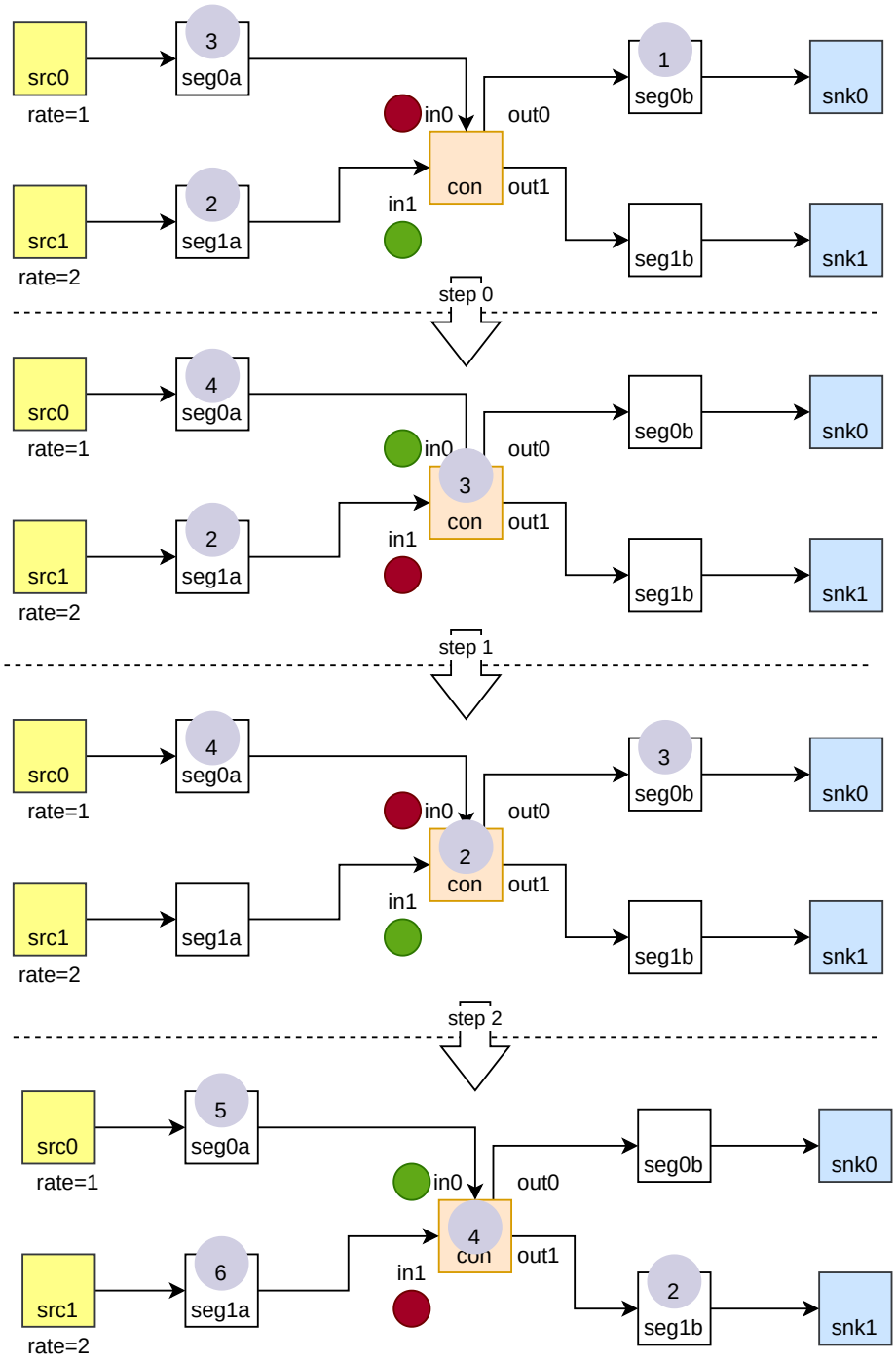

Figure 3: Several consecutive snapshots during the simulation of an example watercraft network. Note that in this figure, watercraft is identified by numbers only for didactive purposes — this should not be a feature of your language.