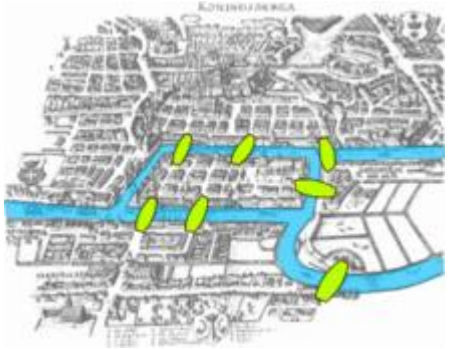
NTNU – Trondheim
Norwegian University of
Science and Technology

Reseda

Networks as a Modeling Tool: a Pedagogical Implementation



by Antoine Rauzy

Licenses & Versions

Reseda is free software distributed by the AltaRica Association under GNU GPLv3 license.

Version	1.0.0
Date	18/02/2021

Table of Contents

1. INTRODUCTION

Rational

Graphs are pervasive in models. Not only virtually all modeling tools use them as internal data structures, but graphs as such are a modeling tool, thanks to fundamental algorithms to determine whether a node B is reachable from a node A, to calculate shortest paths from A to B, to extract strongly connected components, spanning trees, to determine the maximum flow that can circulate from A to B and so on. As a modeling tool, graphs are in general called **Networks**.

Reseda is a pedagogical implementation of (some of) key network algorithms:

- Models are networks written in the **S2ML+NET domain specific modeling language**, which is the combination of S2ML (system structure modeling language), a set of **object-oriented constructs** to structure models and **networks**.
- It implements algorithms.
- It comes as a **command interpreter**, making it possible to perform various studies.

This document specifies S2ML+NET and presents the algorithms implemented by the tool as well as the commands to apply them.

Reseda is developed in Python, for pedagogical purposes only. It is by orders of magnitude less efficient than available commercial tools.

The objective is to familiarize students with network as a **modeling language**.

Installing and Running Reseda

To install Reseda you just need to decompress the archive "Reseda1.0.0.zip" into local directory. Source files are the Python file "Reseda.py" as well as the directory "src" and its content.

To run Reseda, open the file Python file "Reseda.py" into your Python environment, set up the working directory and the name of script file and run the program.

```
36  # 2) Main
37  # -------
38
39  engine = ResedaEngine()
40  engine.OpenSession()
41  engine.SetExecutionStatus(S2MLEngine.ACTIVE)
42  engine.SetCurrentWorkingDirectory("examples/Propagation")
43  engine.LoadScript("propagation.rsd")
44  engine.CloseSession()
```

Organization of this Document

The remainder of this document is organized as follows.

- Section Getting started is a small introduction to Reseda.

The two next sections describe S2ML+NET:

- Section Basic components presents the core of the language.
- Section Structuring constructs presents object-oriented constructs to structure models.
- Section Properties presents the constructs to describe sets of nodes and transitions.

The next section describe the command interpreter:

- Section Commands describes Reseda commands.

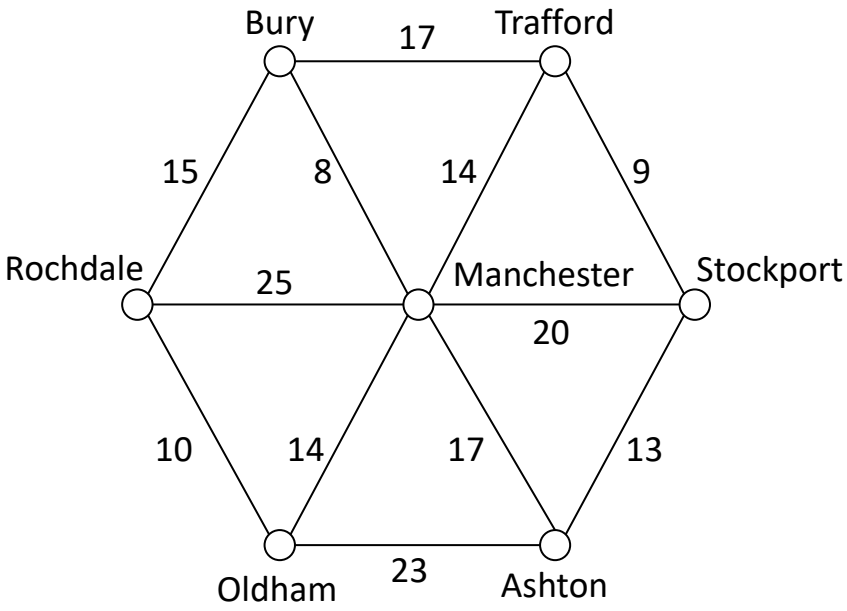
Finally, the appendix completes this document.

- Appendix S2ML+NET gives the Backus-Naur form of the modeling language.
- Appendix Reseda gives the Backus-Naur form of Reseda commands.
- Appendix Known bugs reports know problems with the current version of Reseda.

2. GETTING STARTED

S2ML+NET

S2ML+NET is a textual format to describe (hierarchical) graphs. E.g.




```
block Manchester
    node Bury, Trafford, Rochdale,
        Manchester, Stockport, Oldham,
        Ashton;
    edge
        R01(length=15): Rochdale <-> Bury,
        R02(length=10): Rochdale <-> Oldham,
        R03(length=25): Rochdale <-> Manchester,
        R04(length=8): Bury <-> Manchester,
        R05(length=17): Bury <-> Trafford,
        R06(length=14): Oldham <-> Manchester,
        R07(length=23): Oldham <-> Ashton,
        R08(length=14): Manchester <-> Trafford,
        R09(length=17): Manchester <-> Ashton,
        R10(length=20): Manchester <-> Stockport,
        R11(length=9): Trafford <-> Stockport,
        R12(length=13): Ashton <-> Stockport;
end
```

S2ML+NET (bis)

```
block Manchester
    node Bury,
    …
    edge
        R01(length=15): Rochdale <-> Bury,
        …
end
```

- Each model is described in a **block** which contains declarations of objects of the model. A block starts with keyword **block** followed by the name of the model (here `Manchester`) and ends with the keyword **end**.
- Three types of objects are used to define **graphs**: **nodes**, **edges** and **parameters**. Nodes must be declared before they are referred to in edges.
- Nodes have a name and possibly some attributes.
- Edges have a name and possibly some attributes.
- Edges can be unidirectional, which is indicated with the symbol "->", or bidirectional which is indicated by the symbol "<->".

S2ML+NET (ter)

```
block Manchester
    node Bury,
    …
    edge
        R01(length=15): Rochdale <-> Bury,
        …
end
```

- Attributes are given between after the name (of the node or the edge). They are pairs (name, value), where value is an arithmetic expression, possibly involving parameters.
- Declarations of nodes, edges and parameters are separated with commas “,” and terminated with a semicolon ";".
- Although this is not mandatory, models are usually stored into text files with the extension ".net".

Assessment Process

The assessment process of a model is typically made of the following steps:

1. The model is loaded from a text file.

2. The model is checked and rewritten in a form in which the calculation process can start. This steps is called instantiation in the S2ML jargon.

3. Calculations are performed. Results are printed out into text files.

Scripts

Reseda is a **command interpreter**: it reads commands into a text file and execute them. There are commands to perform each of the steps described in the previous slide.

```
# Step 1: the model is loaded
load "example/Manchester/Manchester.net"

# Step 2: the model is instantiated
instantiate model

# Step 3: some calculations are performed
compute shortest-path Rochdale Stockport length \
    output="result.txt" mode=write
```

Scripts are text files. Although this is not mandatory, models are usually stored into text files with the extension ".rsd".

Results

result.txt

```
Model   Manchester
source  Rochdale
target  Stockport
edge    target  distance
R01 Bury    15
R05 Trafford    32
R11 Stockport   41
```

In result files, items are separated with tabs so that results can be easily loaded into spreadsheets (Excel or equivalent).

3. S2ML+NET:

3.1 Basic Components

Basic Components

Basic components of S2ML+NET models are:

- **Domains** that are sets of symbolic constants;
- **Blocks** that contain declarations of other objects of a model;
- Declarations of **nodes**;
- Declarations of **edges**;
- Declarations of **parameters** involved in arithmetic expressions;
- Declarations of **attributes** associated with nodes and edges;

In the sequel, models are described using `this font`. Keywords are underlined using **`this font`**.

Comments

It is possible to add **comments** everywhere in a S2ML+NET model.

- Any sequence of text between `/*` and `*/` is a comment, even if it spreads over several line.
- All characters after `//` until the end of the line is a comment.

In the sequel, we shall color comments *in italic and green*.

```
/*
 * This is a comment before a block declaration
 */
block Plant // This is a comment till the end of the line
    // declarations
end
```

Identifiers

S2ML+NET models must be written using ASCII characters.

Identifiers, i.e. names of blocks, states, ports and parameters (and other objects introduced in the next sections) obeys the following syntax:

- They start with a letter from a to z or from A to Z.
- They are made of any number of letters, digits, underscores "_".

E.g. `Plant`, `failed`, `R3151`, `this_is_a_valid_although_a_big_long_name`.

Domains

Domains are sets of symbolic constants that can be used in expressions. More exactly, if you want to use a symbolic constant in an expression, you must declare it in a domain. E.g.

```
domain State {STANDBY, WORKING, FAILED}
```

Here the domain `State` contains three symbolic constants `STANDBY`, `WORKING` and `FAILED`.

A symbolic constant may belong to more than one domain.

Blocks

Blocks are the basic container of S2ML+NET. They are **prototypes** in the sense of object-orientation theory. Blocks contain declarations of parameters, states, ports and sources (and other elements that will be described later).

A block declaration starts with the keyword "block", followed by the name of the block. It finishes with keyword "end". E.g.

```
block Valve
    node A, B, C, D;
    parameter Real baseCapacity = 100.0;
    edge
        e1(capacity=0.85 * baseCapacity): A -> B,
        e2(capacity=0.79 * baseCapacity): A -> C,
        e3(capacity=0.83 * baseCapacity): B -> D,
        e4(capacity=0.75 * baseCapacity): C -> D;
end
```

Within a block, all elements must have a different name, even though they are of different types, e.g. a node and a parameter. Elements can be declared in any order.

Nodes & Parameters

Nodes can be declared either individually or several at a time. They can be associated attributes (multiple attributes are separated with commas. E.g.

```
node Bury, Trafford, Rochdale (capacity = 100.0);
```

The three nodes have their capacity set to 100.0.

Parameters are used in arithmetic expressions. They are declared together with the expression that defines them. E.g.

```
parameter Real baseCapacity = 100.0;
parameter Real specialCapacity = 2 * baseCapacity;
```

Edges

Edges have a name and connect two nodes, the **source node** and the **target node**. Both nodes must be declared before the edge is declared.

Edges can be unidirectional, which is indicated with the symbol "->", or bidirectional which is indicated by the symbol "<->".

The declaration of one or several edges must be preceded with the keyword "edge".

Edges are associated attributes. E.g.

```
edge
    e1: A -> B,
    e2: B <-> C,
    e3(capacity = 2.0 * baseCapacity) D -> E;
```

Expressions

Expressions are used to define the value of parameters as well as to define attributes of nodes and edges. The current version of Reseda provides several types of expressions:

- Arithmetic expressions and built-in functions
- Trigonometric functions
- Boolean expressions
- Inequalities
- Random-deviates
- Conditional expressions
- Special built-in functions

In any expression, a reference to a parameter can be used, under the condition that parameters can only depend on parameters and cannot depend on themselves.

Arithmetic Expressions

The current version of Janos implements the following arithmetic expressions.

Syntax	#arguments	Semantics
Floating point number	0	The number
pi	0	π
e1 + … + *en*	≥ 1	Sum of the arguments
e1 - … - *en*	≥ 1	First argument minus the others
e1 * … * *en*	≥ 1	Product of the arguments
e1 / … / *en*	≥ 1	First argument divided by the others
- e	1	Opposite
min(*e1*, …, *en*)	≥ 1	Minimum of its arguments
max(*e1*, …, *en*)	≥ 1	Maximum of its arguments

Examples:

```
0.8 * weight        max(f-g, g-f, 1.0)      3*(x+y)
-e
```

Arithmetic Built-In Functions

The current version of Janos implements the following arithmetic built-ins.

Syntax	#arguments	Semantics
exp(e)	1	exponential
log(*e*)	1	(Natural) logarithm
pow(*e1, e2*)	2	Power
sqrt(*e*)	1	Square root
floor(*e*)	1	Largest integer under
ceil(*e*)	1	Smallest integer above
abs(*e*)	1	Absolute value
mod(*e1, e2*)	2	Modulo
Gamma(*e*)	1	Gamma function

Trigonometric Functions

The current version of Janos implements the following arithmetic built-ins.

Syntax	#arguments	Semantics
sin(*e*)	1	Sine
cos(*e*)	1	Cosine
tan(*e*)	1	Tangent
asin(*e*)	1	Arc sine
acos(*e*)	1	Arc cosine
atan(*e*)	1	Arc tangent

Boolean Expressions

The current version of Janos implements the following Boolean expressions.

Syntax	#arguments	Semantics
`false, true`	0	Boolean constants
e1 **and** … **and** *en*	≥ 1	Conjunction of the arguments
e1 **or** … **or** *en*	≥ 1	Disjunction of the aguments
not e	1	Negation
count(*e1*, …, *en*)	≥ 1	Number of true expressions in the list

Examples:

```
a and b          (f and g) or (not f and h)
not e
```

Inequalities

The current version of Janos accepts the following inequalities.

Syntax	#arguments	Semantics
e1 == *e2*	2	Equal
e1 != *e2*	2	Different
e1 < *e2*	2	Less than
e1 > *e2*	2	Greater than
e1 <= *e2*	2	Less or equal
e1 >= *e2*	2	Greater or equal

Examples:

```
0.8==weight        f - g < x * y
```

Random Deviates

The current version of Janos implements the following random deviates

Syntax	#arg	Semantics
uniformDeviate(*l, h*)	2	The value is drawn at random uniformly in the range [l, h]
normalDeviate(*m, s*)	2	The value is drawn at random according to a normal distribution of mean m and standard deviation s.
lognormalDeviate(*m, s*)	2	The value is drawn at random according to a lognormal distribution of mean m and standard deviation s.
triangularDeviate(*l, h, m*)	3	The value is drawn at random according to a triangular distribution of lower bound l, higher bound h and mode m.

Random Deviates (bis)

The current version of Janos implements the following random deviates

Syntax	#arg	Semantics
`exponentialDeviate(r)`	1	The value is drawn at random according to a the inverse of a negative exponential distribution of rate r.
`WeibullDeviate(a, b)`	2	The value is drawn at random according to the inverse of a Weibull distribution of scale parameter a and shape parameter b.

Conditional Expressions and Special Built-In

The current version of Janos implements the following conditional expressions.

Syntax	#arg	Semantics
`if` *c* `then` *e1* `else` *e2*	3	Equal to e1 if c is true and to e2 otherwise

Example:

```
if x<=y then 1.0 else 2.0
```

The current version of Janos implements the following special built-in.

Syntax	#arg	Semantics
`missionTime()`	0	Returns the current mission time

Priority Rules (Precedence of Operators)

S2ML+DFE (and more generally all languages of the S2ML+X family) obeys the usual precedence rules for operators. Parentheses are used to solve ambiguities.

Priority (decreasing order)
or
and
not
==, !=, <, >, <=, >=
- (n-ary)
+
/
*
- (unary)
all others

```
f and g or not f and h
```
reads
```
(f and g) or ((not f) and h)
```

```
x>=y+0.0 and x<y+1.0
```
reads
```
(x>=(y+0.0)) and (x<(y+1.0))
```

```
3*-x+3/4
```
reads
```
(3*(-x)) + (3/4)
```

Attributes

It is sometimes convenient to define or to redefine an attribute associated with a node or an edge. This can be done by the directive attribute. E.g.

```
attribute MainPropulsionSystem.Converter.state = FAILED;
```

The general form of attribute (re)declaration is as follows.

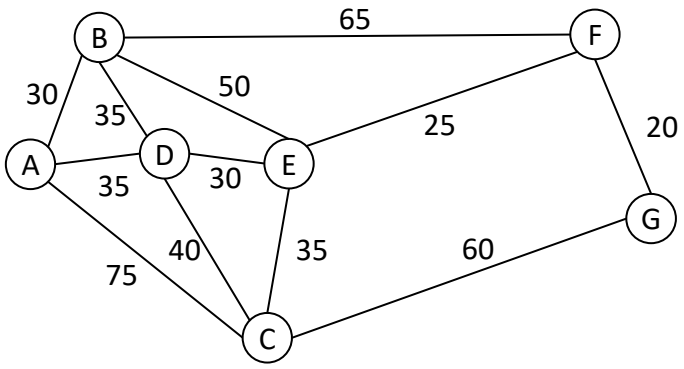
```
attribute Path = Expression;
```

3. S2ML+NET:

3.2 Structuring Constructs

Blocks in Blocks

Consider the graph on the right which describes a railway network. The nodes represent stations and edges are labelled with distances, e.g. in minutes, between these stations.



Trains usually stop at each station they pass through during their journey.
The time they stop at each station must be taken into account to get the duration of a travel.

On way of doing that is to consider stations themselves as (simple) graphs, with one arrival node, one departure node and one edge going from the former to the latter and labelled by the duration of the stop (assuming it is the same for all lines). E.g.

```
block StationA
   node arrival, departure;
   edge stop(time = stopTime): arrival -> departure;
   parameter Real stopTime = 3;
end
```

Blocks in Blocks (bis)

The system can then be represented by the following hierarchical model:

```
block Network
   block StationA
      …
   end
   block StationB
      …
   end
   …
   edge
      AtoB(time=30): A.departure -> B.arrival;
      BtoA(time=30): B.departure -> A.arrival;
      …
end
```

The block `Network` **composes** the blocks `StationA` and `StationB`.

Instantiated Form

The previous hierarchical model is equivalent to the following instantiated model:

```
block Network
   node StationA.arrival, StationA.departure;
   edge
      A.stop(time = A.stopTime): A.arrival -> A.departure;
   parameter Real A.stopTime = 3;
   node StationB.arrival, StationB.departure;
   edge
      B.stop(time = B.stopTime): B.arrival -> B.departure;
   parameter Real B.stopTime = 3;
   …
   edge
      AtoB(time=30): A.departure -> B.arrival;
      BtoA(time=30): B.departure -> A.arrival;
      …
end
```

Cloning

Duplicating "by hand" blocks representing similar components would be both tedious and error prone in large systems studies. **Cloning** is the a first solution to this problem.

```
block Network
   block StationA
      node arrival, departure;
      edge stop(time = stopTime): arrival -> departure;
      parameter Real stopTime = 3;
   end
   clones StationA as StationB;
   …
   edge
      AtoB(time=30): A.departure -> B.arrival;
      BtoA(time=30): B.departure -> A.arrival;
      …
end
```

This model is equivalent to the first one: their instantiated form are the same.

Models as Scripts

It is possible to change elements of clones in two ways.
Either directly in the clone directive:

```
clones StationA as StationB
   parameter Real stopTime = 5.0;
   end
```

Or later in the model:

```
clones StationA as StationB;
parameter Real StationB.stopTime = 5.0;
```

This results of the "**model as script**" concept.

Paths

Thanks to absolute and relative paths, it is possible to refer to any element from any block of hierarchical model.

```
block Network
  block Zone1
    block StationA
      node arrival, departure;
    end
  end
  block Zone2
    block StationB
      node arrival, departure;
      edge
        ToStationA: departure -> main.Zone1.StationA.arrival;
        FromStationA: owner.owner.Zone1.StationA.departure -> arrival;
    end
  end
end
```

Absolute path: the keyword **main** denotes the top level block of the hierarchy.

Relative path: the keyword **owner** denotes the parent block in the hierarchy.

Classes & Instances

Another solution consists in creating a on-the-shelf reusable component, via the notion of class. Then to instantiate this component as many time as needed. This makes it possible to create libraries of reusable modeling components.

```
class Station
   node arrival, departure;
   edge stop(time = stopTime): arrival -> departure;
   parameter Real stopTime = 3;
end

block System
   Station A;
   Station B;
   …
   edge
      AtoB(time=30): A.departure -> B.arrival;
      BtoA(time=30): B.departure -> A.arrival;
      …
end
```

Again, this model is equivalent to the first one: their instantiated form are the same.

Models as Scripts (bis)

It is possible to change elements of instances in two ways.
Either directly in the instance declaration:

```
Station StationB
   parameter Real stopTime = 5.0;
   end
```

Or later in the model:

```
Station StationB;
parameter Real StationB.stopTime = 5.0;
```

Inheritance

Assume that we have to consider two kinds of stations: simple stations, which only one duration of stops and large stations, with two (or more) stop durations. It is possible to create first a class describing simple stations, then to extends this class for large stations. This mechanism is called **inheritance**.

```
class SimpleStation
   node arrival, departure;
   edge stop(time = stopTime): arrival -> departure;
   parameter Real stopTime = 3;
end

class LargeStation
   extends SimpleStation;
   node largeArrival, largeDeparture;
   edge
      largeStop(time = largeStopTime):
         largeArrival -> largeDeparture;
   parameter Real largeStopTime = 5;
end
```

Aggregation

Aggregation denotes a "uses" kind of a relation between blocks (or instances of classes).

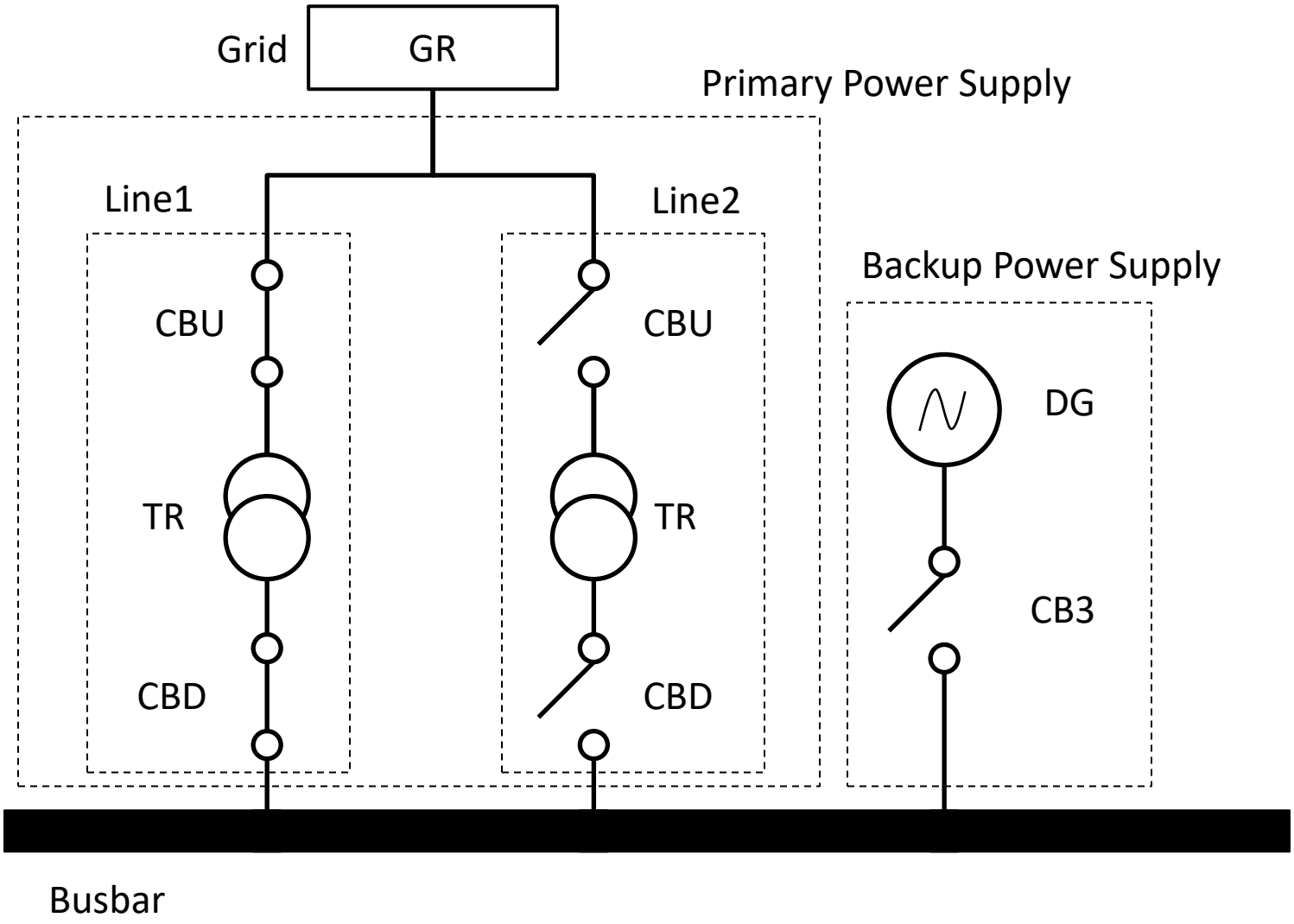
```
block Network
    block Controller
        parameter Real shortStopTime = 3.0;
    end
    block Zone1
        block StationA
            embeds main.Controller as CTRL;
            node arrival, departure;
            edge
                shortStop(time = CTRL.shortStopTime)
                arrival -> departure;
        end
    end
end
```

Aggregation: within the block `StationA`, `CTRL` becomes an alias for **`main`**`.Controller`.

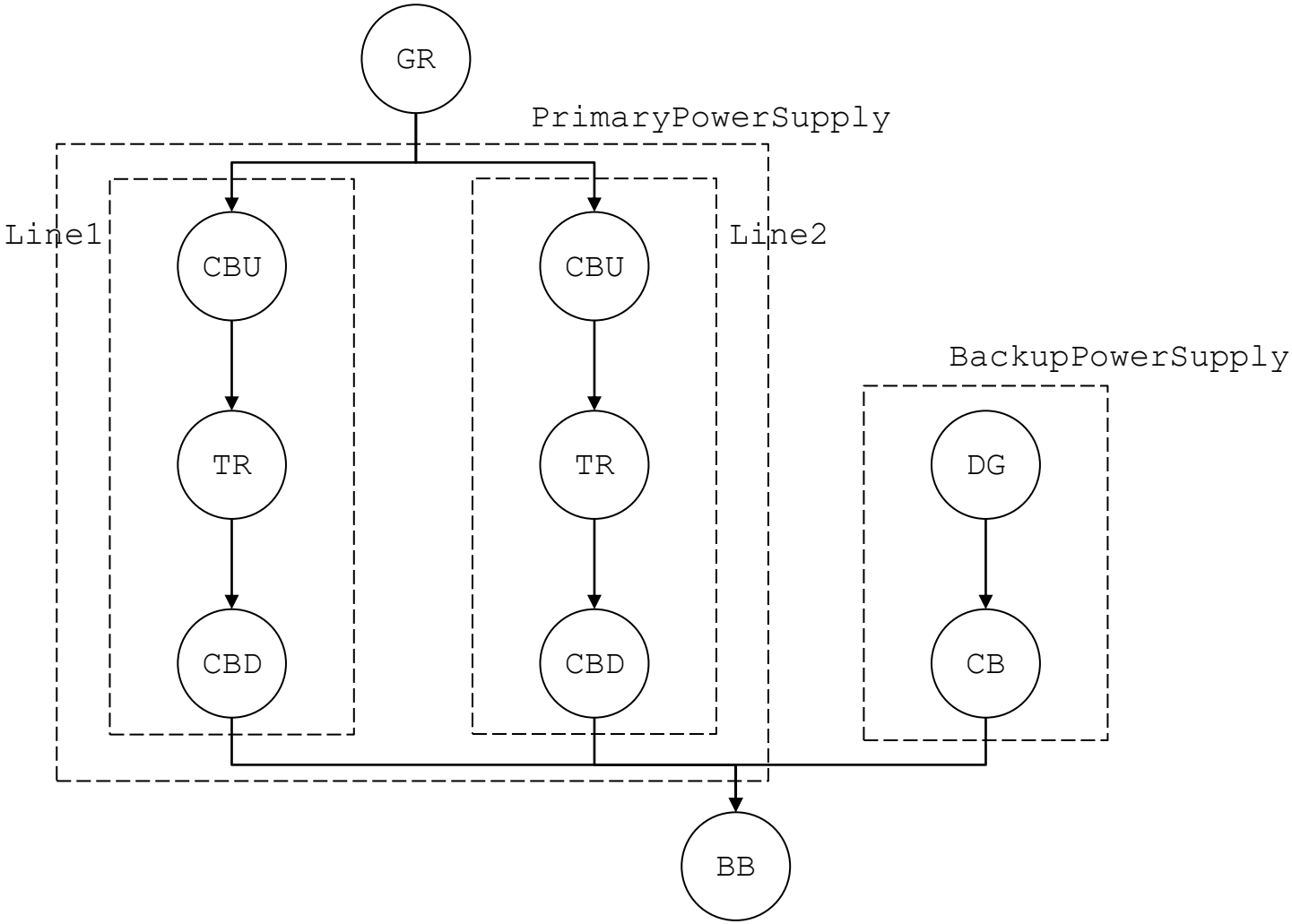
3. S2ML+NET:

3.3 PROPERTIES

Offsite Power Supply: P&ID



Offsite Power Supply: Network



```
block OffsidePowerSupply
   node GR(active = true);
   block PrimaryPowerSupply
      block Line1
         node CBU(active = true);
         node TR(active = true);
         node CBD(active = true);
         edge cbu-tr: CBU -> TR;
         edge tr-cbd: TR -> CBD;
      end
      clones Line1 as Line2
         attribute CBU.active = false;
         attribute TR.active = false;
         attribute CBD.active = false;
   end
   block
      node DG(active = false);
      node CB(active = false);
      edge dg-cb: DG -> CB;
   end
   node BB(active = true);
   …
end
```

```
block OffsidePowerSupply
   ...
   edge gr-cbu1: GR -> PrimaryPowerSupply.Line1.CBU;
   edge gr-cbu2: GR -> PrimaryPowerSupply.Line2.CBU;
   edge cbd1-bb: PrimaryPowerSupply.Line1.CBD -> BB;
   edge cbd2-bb: PrimaryPowerSupply.Line2.CBD -> BB;
   edge cbdg-bb: BackupPowerSupply.CB -> BB;

   assertion
      Set PowerSources = {GR, BackupPowerSupply.DG};
      Set PoweredNodes =
         reachableNodes(PowerSources, active, true);
      Set PoweredBusbars = PoweredNodes /\ { BB };

end
```

Assertions and Sets

Reseda makes it possible to describe **sets** of blocks, nodes and edges. These sets can be described extensionally, by giving their elements, or intentionally by characterized them in a logical way.
Sets are declared in an **assertion**. Assertions defined in any block (and object-oriented constructs instantiated in blocks).

```
block OffsidePowerSupply
   ...
   assertion
      Set PowerSources = {GR, BackupPowerSupply.DG};
      Set PoweredNodes =
         reachableNodes(PowerSources, active, true);
      Set PoweredBusbars = PoweredNodes /\ { BB };

end
```

extensional definition

intensional definition

Set Expressions: Extensional Sets and Set Operations

Extensional sets:

```
{GR, BackupPowerSupply.DG}
```

Set Operations

Syntax	#arguments	Semantics
S1 \/ … \/ *Sn*	≥ 1	Union
S1 /\ … /\ *Sn*	≥ 1	Intersection
S1 / *S2*	2	Set difference
S : *E*	1 : 1	Selection on the attribute values E: Boolean expression

Builtin Set Expressions (1)

Syntax	#arg	Semantics
parentBlocks(S)	1	Set of parent blocks of elements given in argument
ancestorBlocks(S)	1	Set of ancestor blocks of elements given in argument
childBlocks(S)	1	Set of child blocks of blocks given in argument
descendantBlocks(S)	1	Set of descendant blocks of blocks given in argument
childNodes(S)	1	Set of child blocks of nodes given in argument
descendantNodes(S)	1	Set of descendant nodes of blocks given in argument
childEdges(S)	1	Set of child edges of nodes given in argument
descendantEdges(S)	1	Set of descendant edges of blocks given in argument
inEdges(S)	1	Set of in-coming edges of nodes given in argument
outEdges(S)	1	Set of out-going edges of blocks given in argument

Builtin Set Expressions (2)

Syntax	#arg	Semantics
`reachableNodes(S, E, F)`	3	Set of nodes that are reachable from nodes given in first argument. The second argument is a Boolean expression that tests the attributes of the nodes. The third argument is a Boolean expression that tests the attributes of the edges.
`coreachableNodes(S, E, F)`	3	Set of nodes from nodes which nodes given in first argument are reachable. The second argument is a Boolean expression that tests the attributes of the nodes. The third argument is a Boolean expression that tests the attributes of the edges.

4. RESEDA COMMANDS

Role of Commands

Categories of commands:

Reseda commands can be split into the following categories.

1. Commands to load, to check and to instantiate models.
2. Commands to compute indicators.
3. Commands to print out information, e.g. models.

Normally, commands of type 1, 2 and 3 are applied in sequence.

Order of arguments:

The order of arguments in a command matters, even though some arguments are optional. You have to follow strictly the syntax described in this section.

Optional arguments:

Optional arguments are given in a special form: `name=value`, where `name` is the name of the argument and `value` its value.

General Considerations

One line commands:

Reseda commands spread normally over one line. It is however possible to write a command on several line by escaping the end of line is escaped with an anti-slash "\", e.g.

```
compute shortest-path departure arrival time \
    output="statistics.txt" mode=append
```

Comments:

Comments can be introduced in Reseda scripts. Any character between the character # and the end of the line is a comment. We shall underline comment in green. E.g.

```
# this is a comment
```

File Names and Modes

File names:

Most of the commands require an input or an output file name as argument. File names can be given directly, e.g. `examples/Manchester.net` or surrounded with quotes, e.g. `"examples/Manchester.net"`.

The second form is mandatory when the file name or path includes spaces. It is wise to use it anyway.

Output file modes:

When opening a file to print out something, Reseda can do it in two modes: either the file is overwritten if it exists already -- this is the mode `write` --, or the new information is appended at the end of the existing file -- this is the mode `append`. If the file did not exist, it is created in both cases. By default, the mode is `write`. E.g.

```
compute shortest-path A B time output="path.txt" mode=append
```

The above command calculate the shortest path from node `A` to node `B` and appends the result to the file `probas.txt`.

Result Files

As much as possible, result files are organized in a way they can be loaded into spreadsheets (Excel or equivalent). Items are separated with tabs. Methods to load such text files differ from one spreadsheet tool to another.

result.txt

```
Model   Stations
source  A.departure
target  G.arrival
edge    target  distance
AtoB    B.arrival   30
B.stop  B.departure 40
BtoF    F.arrival   105
F.stop  F.departure 115
FtoG    G.arrival   135
```

Commands to Compute Indicators

`compute shortest-path` *`sourceNodeName targetNodeName attributeName options`*

This command applies the Dijskra's algorithm to compute the shortest path from the given source node to the given target node. Length of edges are calculated from the given attribute. The result is stored in the given file.

Lengths of edges must be non-negative.

`compute dates` *`sourceNodeName targetNodeName attributeName`* `options`

This command calculate the early dates and the last dates (as in PERT diagrams) of nodes of the given graph, start the given source node and ending in the given target node. Duration of activities (edges) are calculated from the given attribute. The result is stored in the given file.

The graph must be oriented and loop-free. Durations of edges must be non-negative.

Commands to Compute Indicators (bis)

`compute maximum-flow` *`sourceNodeName targetNodeName attributeName options`*

This command applies the Ford and Fulkerson algorithm to compute the maximum flow from the given source node to the given target node. Capacity of edges are calculated from the given attribute. The result is stored in the given file.

Capacity of edges must be non-negative.

Commands to Compute Sets

`compute properties` *`options`*

This command computes the values of sets declared in assertion.

Options are the following.

- `target-block = blockName`
- `output = filename`
- `mode = write or append`
- `mission-time = [ number ("," Number)* ]`

Commands to Print Information

`print model` *`fileName`* `[mode=(append|write)]`

This command prints out the original model.

`print instantiated-model` *`fileName`* `[mode=(append|write)]`

This command prints out the instantiated model.

5. REFERENCES

References

Recommend books on (graphs) algorithms:

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms (Second ed.). The MIT Press. Cambridge, MA, USA. ISBN ISBN 0-262-03293-7. 2001.

Robert Endre Tarjan. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics. Philadelphia, PA, USA. ISBN 9780898711875. 1983.

APPENDIX

APPENDIX
A. GRAMMAR OF S2ML+NET

Models

```
Model ::= (DomainDeclaration | ClassDeclaration | BlockDeclaration)*

DomainDeclaration ::=
    domain Identifier "{" Identifier ("," Identifier)* "}"

ClassDeclaration ::=
    class Identifier BlockField* end

BlockDeclaration ::=
    block Identifier BlockField* end

BlockField ::=
        NodeDeclaration | EdgeDeclaration| ParameterDeclaration
    |   BlockDeclaration | InstanceDeclaration
    |   ExtendsDirective | EmbedsDirective | ClonesDirective
    |   AttributeDirective
    |   Assertion
```

Nodes, Edges & Parameters

```
NodeDeclaration ::=
    node Identifier Attributes? ";"

EdgeDeclaration ::=
    edge Identifier Attributes? ":" Path ("->" | "<->") Path ";"

Attributes ::=
    "(" Attribute ( "," Attribute )+ ")"
Attribute ::=
    Identifier "=" Expression

ParameterDeclaration ::=
    parameter DomainIdentifier Path "=" Expression ";"
```

Expressions

```
Expression ::=
        Path
    |   ArithmeticExpression | ArithmeticBuiltIn | TrigonometricFunction
    |   BooleanExpression | Inequality
    |   RandomDeviate
    |   ConditionalExpression | SpecialBuiltIn
    |   "(" Expression ")"


ConditionalExpression ::=
    if BooleanExpression then Expression else Expression

SpecialBuiltIn ::=
    missionTime()
```

Arithmetic Expressions

```
ArithmeticExpression ::=
        Float
    |   pi
    |   ArithmeticExpression ( "+" ArithmeticExpression )+
    |   ArithmeticExpression ( "-" ArithmeticExpression )+
    |   ArithmeticExpression ( "*" ArithmeticExpression )+
    |   ArithmeticExpression ( "/" ArithmeticExpression )+
    |   "-"  ArithmeticExpression
    |   min "(" ArithmeticExpression ( "," ArithmeticExpression )*  ")"
    |   max "(" ArithmeticExpression ( "," ArithmeticExpression )* ")"
```

Arithmetic Built-In & Trigonometric Functions

```
ArithmeticBuiltIn ::=
      |    exp "(" ArithmeticExpression ")"
      |    log "(" ArithmeticExpression ")"
      |    pow "(" ArithmeticExpression "," ArithmeticExpression ")"
      |    sqrt "(" ArithmeticExpression ")"
      |    floor "(" ArithmeticExpression ")"
      |    ceil "(" ArithmeticExpression ")"
      |    abs "(" ArithmeticExpression ")"
      |    mod "(" ArithmeticExpression "," ArithmeticExpression ")"
      |    Gamma "(" ArithmeticExpression ")"

TrigonometricFunction ::=
      |    sin "(" ArithmeticExpression ")"
      |    cos "(" ArithmeticExpression ")"
      |    tan "(" ArithmeticExpression ")"
      |    asin "(" ArithmeticExpression ")"
      |    acos "(" ArithmeticExpression ")"
      |    atan "(" ArithmeticExpression ")"
```

Boolean Expressions & Inequalities

```
BooleanExpression ::=
    |   BooleanExpression ( or BooleanExpression )+
    |   BooleanExpression ( and BooleanExpression )+
    |   not BooleanExpression
    |   count "(" BooleanExpression ( "," BooleanExpression )* ")"

Inequality ::=
    |   Expression "==" Expression
    |   Expression "!=" Expression
    |   ArithmeticExpression "<" ArithmeticExpression
    |   ArithmeticExpression ">" ArithmeticExpression
    |   ArithmeticExpression "<=" ArithmeticExpression
    |   ArithmeticExpression ">=" ArithmeticExpression
```

Random Deviates

```
RandomDeviate ::=
        uniformDeviate "(" [ArithmeticExpression]2 ")"
    |   normalDeviate "(" [ArithmeticExpression]2 ")"
    |   lognormalDeviate "(" [ArithmeticExpression]2 ")"
    |   triangularDeviate "(" [ArithmeticExpression]3 ")"
    |   exponentialDeviate "(" ArithmeticExpression ")"
    |   WeibullDeviate "(" [ArithmeticExpression]2 ")"
```

Directives

```
InstanceDeclaration ::=
        Identifier Identifier ";"                   # ClassName InstanceName
    |   Identifier Identifier BlockField* end       # idem

ClonesDirective ::=
        clones Path as Identifier ";"               # BlockPath CloneName
    |   clones Path as Identifier BlockField* end   # idem

ExtendsDirective ::=
        extends Identifier ";"                      # ClassName

EmbdesDirective ::=
        embeds Path as Identifier ";"               # BlockPath LocalName
    |   embeds Path as Identifier BlockField* end   # idem

AttributeDirective ::=
    attribute Path "=" Expression ";"
```

Identifiers, Paths, Constants & Comments

```
Identifier ::= [a-zA-Z_][a-zA-Z0-9_-]+

Path ::= Identifier ( "." Identifier )*

Integer ::= [0-9]+

Float ::= [+-]? [0-9]+ ("." [0-9]+)? ([eE] [+-]? [0-9]+)?
```

Comments can be added everywhere in the code.

- Single line comments introduced by `//`, which comment out the rest of the line.
- Multiline comments which comment out the text between `/*` and `*/`.

Assertion

```
Assertion ::=
    assertion SetDeclaration

SetDeclaration ::=
    Set Path "=" SetExpression ";"

SetExpression ::=
        Path
    |   "{" Path ( "," Path )* "}"
    |   BuiltinSetExpression
    |   SetExpression ( "\/" SetExpression )*
    |   SetExpression ( "/\" SetExpression )*
    |   SetExpression "/" SetExpression
    |   SetExpression ":" TestOnAttributes
    |   "(" SetExpression ")"
```

TestOnAttributes ::=

BooleanExpression

Builtin Set Expressions

```
BuiltInSetExpression ::=
        parentBlocks "(" SetExpression ")"
    |   ancestorBlocks "(" SetExpression ")"
    |   childBlocks "(" SetExpression ")"
    |   descendantBlocks "(" SetExpression ")"
    |   childNodes "(" SetExpression ")"
    |   descendantNodes "(" SetExpression ")"
    |   childEdges "(" SetExpression ")"
    |   descendantEdges "(" SetExpression ")"
    |   sourceNodes "(" SetExpression ")"
    |   targetNodes "(" SetExpression ")"
    |   inEdges "(" SetExpression ")"
    |   outEdges "(" SetExpression ")"
    |   reachableNodes "(" SetExpression ","
            TestOnAttributes "," TestOnAttributes  ")"
    |   coreachableNodes "(" SetExpression ","
            TestOnAttributes "," TestOnAttributes  ")"
```

APPENDIX
B. GRAMMAR OF RESEDA COMMANDS

Scripts, Commands load and instantiate

```
Script ::= Command*

Command ::=
        CommandLoad
    |   CommandInstantiate
    |   CommandCompute
    |   CommandPrint
    |   CommandSet

CommandLoad ::=
        load model fileName
    |   load script fileName

CommandInstantiate ::=
    instantiate model
```

Command compute

```
CommandCompute ::=
        CommandComputeShortestPath
    |   CommandComputeDates
    |   CommandComputeMaximumFlow
    |   CommandComputeProperties

CommandComputeShortestPath ::=
    compute shortest-path sourceNodeName targetNodeName attributeName
        CommandComputeOption*

CommandComputeDates ::=
    compute dates sourceNodeName targetNodeName attributeName
        CommandComputeOption*

CommandComputeMaximumFlow ::=
    compute maximum-flow sourceNodeName targetNodeName attributeName
        CommandComputeOption*
```

Command compute (bis)

```
CommandComputeProperties ::=
     compute properties CommandComputeOption*

CommandComputeOption ::=
          target-block = blockname
     |    output = filename
     |    mode = (append|write)
     |    mission-time = "[" Number ("," Number)* "]"
```

Command print

```
CommandPrint ::=
        CommandPrintModel
    |   CommandPrintInstantiatedModel

CommandPrintModel ::=
    print model fileName [mode=(append|write)]

CommandPrintInstantiatedModel ::=
    print instantiated-model fileName [mode=(append|write)]
```

Command set

```
CommandSet ::=
     set option-name value
```

Comments

All characters comprised between a # symbol and the end of the line are considered as part of a comment.

Appendix
C. Known Bugs

Known Bugs

- No error message for edges with no capacity.