TESLA PARSER: A BRIEF INTRODUCTION

The general structure of a rule is as follows (this is Rule R1 from this paper http://home.deib.polimi.it/cugola/Papers/trex.pdf), adapted so that it can be parsed from the Java client:

```
define Fire(area: string, measuredTemp: double)
from Smoke(area=>$a) and
each Temp([string]area=$a, value>45)
within 300000 from Smoke
where area:=Smoke.area, measuredTemp:=Temp.value
```

NAMING / OPERATORS CONVENTIONS

- The name of a predicate begins with a capital letter and is followed by lowercase letters.
- The name of an attribute is composed only by lowercase letters.
- The size of the windows is implicitly expressed in milliseconds.
- A colon ‘ : ‘ is used between the names of the attributes of the complex event and their value types.
- The operator ‘ := ‘ is used to specify the value of the attributes of the complex event
- The operator ‘ => ‘ is used to rename an attribute of an event (more on this in the parameters section)
- The names of the aggregate functions are (AVG, MAX, MIN, SUM, COUNT) and must written in capital letters
- Selection policies must be expressed with (‘last’, ‘first’, ‘each’) operators, written with lowercase letters
- Supported binary operators for computation are (‘+’, ‘-’, ‘*’, ‘/’, ‘&’, ‘|’)
- Supported binary operators for comparison are (‘>’, ‘<’, ‘=’, ‘!=’)
- Even if a predicate/negation/aggregate has no parameter/constraints, you must still write the opening/closing brackets after its name
- The rule ends with a semicolon ‘ ; ‘

VALUE TYPES:

The parser automatically detects the type of a value, that can be int, float, bool or string.
Float values always have a dot; so if you want to express an integer as a float, write it as X.0
Boolean values can be expressed only as ‘true’ or ‘false’.
String values are expressed by quotation marks.

STATIC ATTRIBUTES/CONSTRAINTS

For a static assignment to be computed as static, you must express it with a single value; as example “where value := 1 + 3 * 2” won’t be parsed as a static assignment, and the generated

rule will be more complex for no reason. Instead “where value := 7” wil be treated as static, and won’t require any complex computation to the engine.

PARAMETERS:
A parameter must always begin with the value type that drives the comparison, expressed within square parentheses. The values of the attributes in the expression of a parameter will be casted to that value type.

Also parameters must always be written so that on the left side of the expression there is only the reference to the attribute of the predicate being specified; as example

“value * 2 > $a” would not be parsed correctly, while “value > $a / 2” is good.

References to attributes of different events can be expressed in two ways: you can tag an attribute with another name with the ‘ => ‘ operator followed by a new name that begins with ‘ $ ’ and then use the new name, or else you can simply refer to them as Predname.Attrname; as example the following rules are equal:

define Ce(area: string, measuredtemp: float) from Smoke(value => $a) and last Rain() within 500000 from Smoke and last Temp([int]value > $a * 2 + 17) within 500000 from Smoke where area := "foo", measuredtemp:=AVG(Rain.value([float]value > $a /2)) between Smoke and Temp;

define Ce(area: string, measuredtemp: float) from Smoke() and last Rain() within 500000 from Smoke and last Temp([int]value > Smoke.value * 2 + 17) within 500000 from Smoke where area := "foo", measuredtemp:=AVG(Rain.value([float]value > Smoke.value /2)) between Smoke and Temp;

NEGATIONS:
Negations can be expressed just like parameters, but they must begin with the ‘not’ keyword and they can be defined to checked onyl between two predicates with the ‘between’ keyword:

Neg1:
define Ce(area: string, measuredtemp: float) from Smoke(value => $a) and last Rain() within 500000 from Smoke and not Temp([int]value > $a * 2 + 17) between Smoke and Rain where area := "foo", measuredtemp:=AVG(Rain.value([float]value > $a /2)) between Smoke and Temp;

Neg2;
define Ce(area: string, measuredtemp: float) from Smoke(value => $a) and last Rain() within 500000 from Smoke and not Temp([int]value > $a * 2 + 17) within 100000 from Smoke where area := "foo", measuredtemp:=AVG(Rain.value([float]value > $a /2)) between Smoke and Temp;

AGGREGATES:
Aggregates can be expressed with the name of the corresponding function, followed by the Name.attribute couple that specifies the values that must be used for the computation, then by the predicates and finally by the selection policy (that is just like that of a negation).

CONSUMING:
Consuming can be expressed with the ‘ consuming ‘ keyword followed by a comma separated list of predicate names; as example:

define Ce(area: string, measuredtemp: float) from Smoke(value => $a) and last Rain() within 500000 from Smoke and last Temp([int]value > $a * 2 + 17) within 500000 from Smoke where area := "foo", measuredtemp:=AVG(Rain.value([float]value > $a /2)) between Smoke and Temp consuming Rain, Temp;