RobMoSys

RoQME

RoQME Modeling Language

A quick glance

Software Engineering Group
QUERCUS

UNIVERSIDAD DE EXTREMADURA

AVISPA

UNIVERSIDAD DE MÁLAGA

BIOMETRIC VOX

Property

- Output variable to represent the degree of fulfilment of a non-functional property.

(Basic syntax) `property NAME [reference VALUE] [survival TIME]`

defines the reference value (a number between 0 and 1), i.e., the value that is reached without any evidence. By default, it is 0.5

The influence of observations becomes irrelevant as time passes. This option defines the average survival time of this influence

(Example) `property effectiveness reference 1 survival 30min`

Context variables

- Variables representing context inputs.

(Basic syntax) `context NAME : DATATYPE`

(Example) `context alarmConnected : boolean`

Possible data types:

`enum, boolean, eventtype, number`

And arrays, e.g.: `number[4], boolean[N]`

- Context variables can also be used to derive complex context from primitive inputs.

(Syntax) `context NAME : DATATYPE := EXPRESSION`

(Example)

```
context motionDetected : eventtype

context motion : enum {MOTIONLESS, MOVING}
          := count(motionDetected, 1min)>5 ?
          motion::MOVING : motion::MOTIONLESS
```

Observations

- Statements for specifying relevant context patterns and their influence on QoS properties.

(Basic syntax)

```
observation NAME : PATTERN EXPRESSION
    (reinforces | undermines) PROPERTY [verylow | low | high | veryhigh]
```

Influence of the observation

Strength of the observation

(Example)

```
observation obs1 : !collision reinforces effectiveness HIGH

observation obs2 : motion > 10 undermines safety
```

Pattern expressions: update

`update(expr)`

Input: expression of any type

Output: produces an event when any variable in *expr* is updated and, by extension, *expr* is also updated

Examples:

```
context ctx1 : number

context ctx2 : boolean

observation o1 : update(ctx1)

observation o2 : update(ctx1 > 45)

observation o3 : update(ctx1 > 45 | ctx2)
```

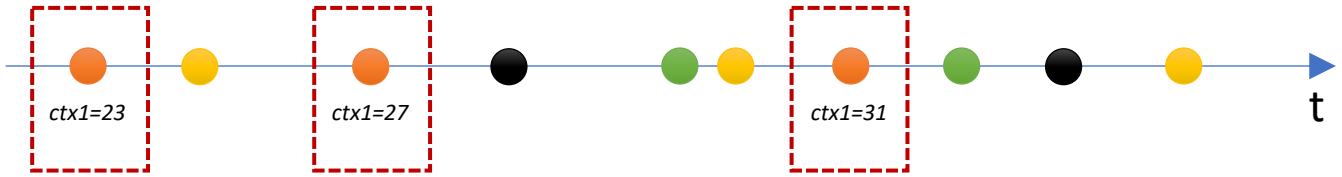
o1 is satisfied when ctx1 receives an update

o2 is satisfied when the value of ctx1>45 changes

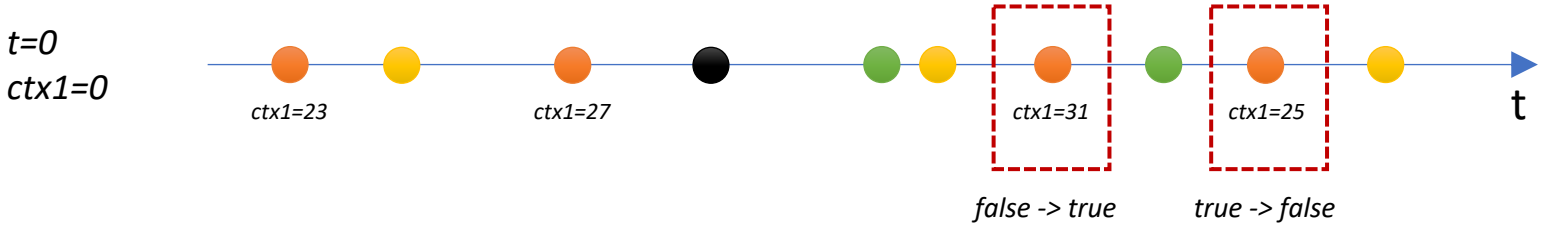
o3 is satisfied when the value of (ctx1>45 or ctx2) changes

Pattern expressions: update

```
observation o1 : update(ctx1)
```




```
observation o1 : update(ctx1>30)
```



Pattern expressions: eventWhen

eventWhen(*expr*)

Input: Boolean expression

Output: produces an event when update(*expr*) and *expr* = true

Examples:

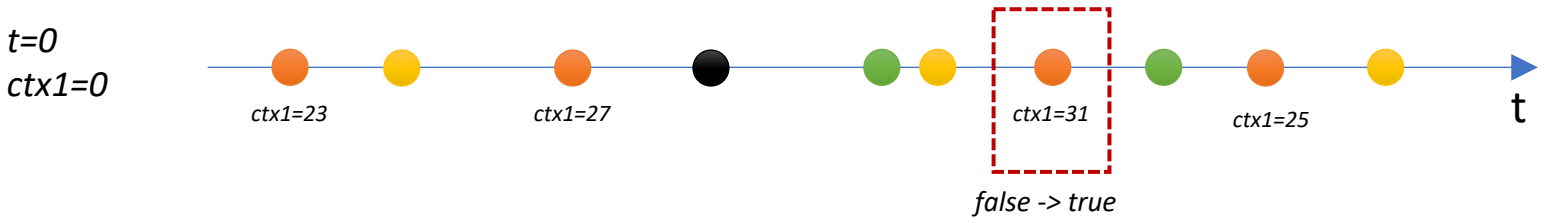
```
context ctx1 : number

context ctx2 : boolean

observation o1 : eventWhen(ctx1 > 45)

observation o2 : eventWhen(ctx2)
```

o1 is satisfied when the value of ctx1>45 changes to true

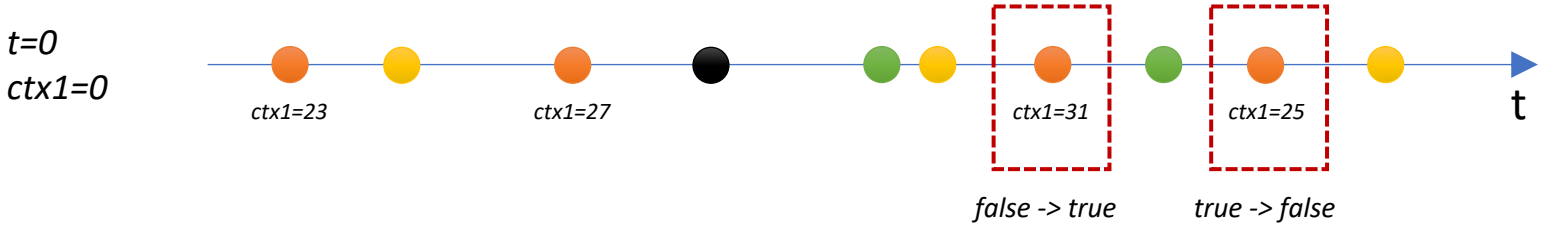
o2 is satisfied when the value of ctx2 changes to true

Pattern expressions: eventWhen

```
observation o1 : eventWhen(ctx1>30)
```




```
observation o1 : update(ctx1>30)
```



Pattern expressions: expr

expr

Input: expression of any type

Output: produces an event when: eventWhen(*expr*) if *expr* is a Boolean expression, otherwise, update(*expr*).

Therefore, depending on the data type of the expression, the patterns *eventWhen* and *update* are implicit

Examples:

```
context ctx1 : number

context ctx2 : boolean

observation o1 : ctx1 > 45

observation o2 : ctx1

observation o3 : ctx2
```

It is equivalent to eventWhen(ctx1>5)

It is equivalent to update(ctx1)

It is equivalent to eventWhen(ctx2)

Pattern expressions: and

expr1 **and** *expr2*

Input: event-typed expressions

Output: produces an event when both expressions occurred

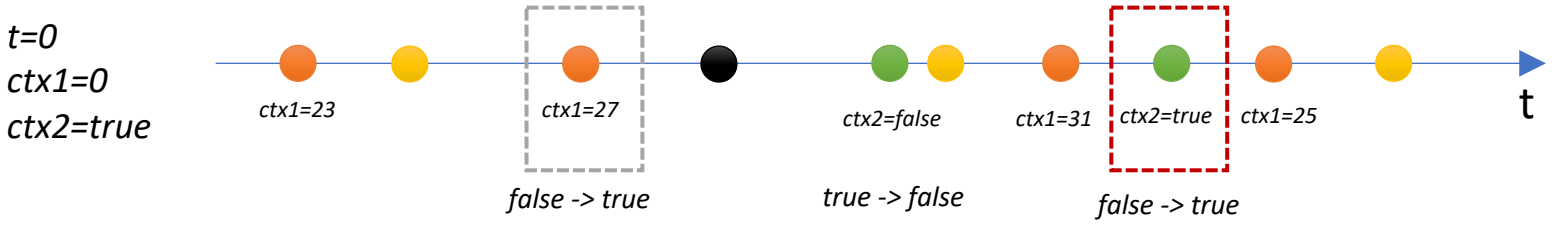
Examples:

```
context ctx1 : number

context ctx2 : boolean

observation o1 : ctx1 > 25 and ctx2

observation o2 : ctx1 > 25 & ctx2
```

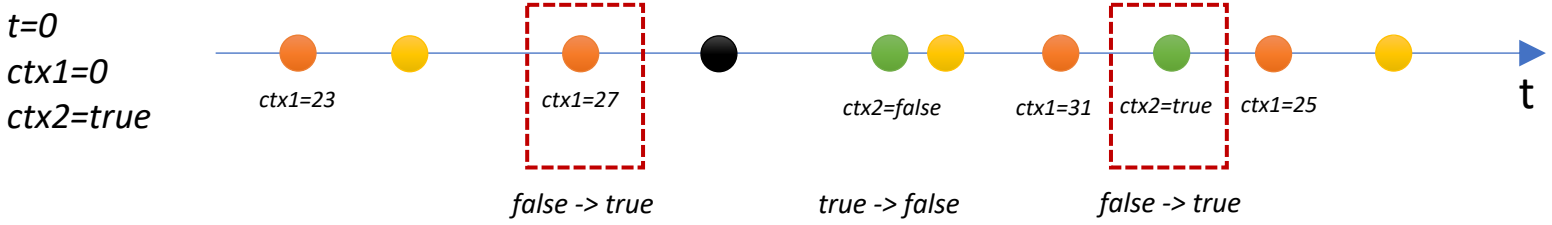
RoQME considers two AND operators: the logical (&) and the pattern-based one (and). While in the first one two events are necessary, e.g.: eventWhen(ctx>25) and eventWhen(ctx2), in the second one, we just need one update and the condition evaluated to true, i.e., eventWhen(ctx>25 & ctx2)

Pattern expressions: and

```
observation o2 : ctx1 > 25 & ctx2
```




```
observation o1 : ctx1 > 25 and ctx2
```



Pattern expressions: or

expr1 **or** *expr2*

Input: event-typed expressions

Output: produces an event when one of the expressions occurred

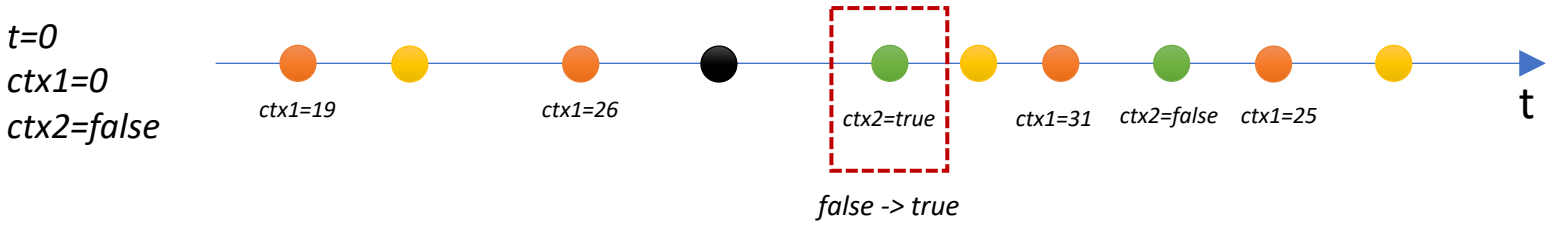
Examples:

```
context ctx1 : number

context ctx2 : boolean

observation o1 : ctx1 > 25 or ctx2

observation o2 : ctx1 > 25 | ctx2
```

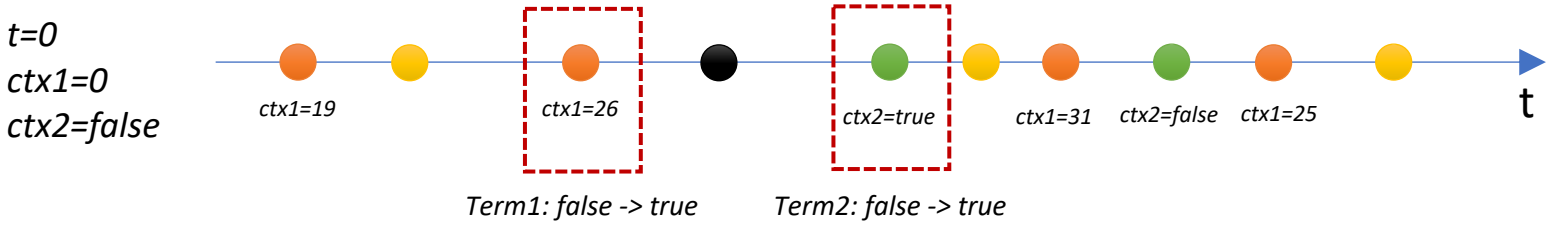
RoQME considers two OR operators: the logical (|) and the pattern-based one (or). Therefore o1 and o2 does not have the same behavior. o2 is satisfied when one of the variables is updated and the global expression (ctx>25 | ctx2) changes its value to true. On the contrary, o1 can be satisfied if ctx1>25 changes to true independently of ctx2

Pattern expressions: or

```
observation o2 : ctx1 > 25 | ctx2
```




```
observation o1 : ctx1 > 25 or ctx2
```



Pattern expressions: followed-by

expr1 -> expr2

Inputs: event-typed expressions

Output: produces an event when *expr1* is met after *expr2*

Examples:

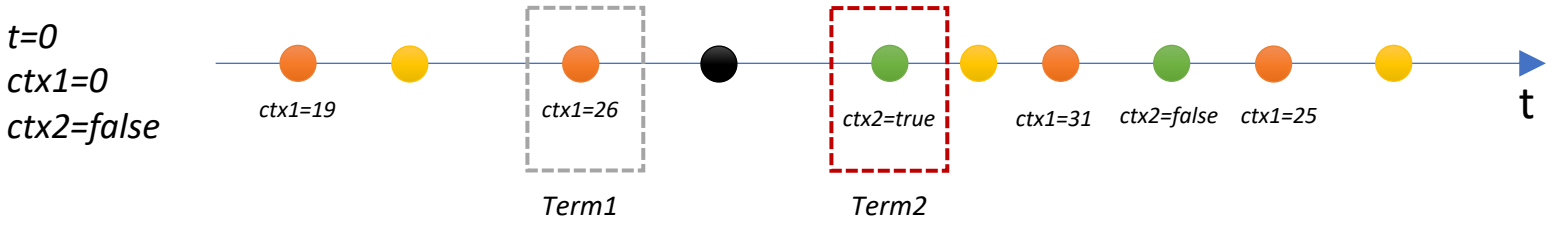
```
context ctx1 : number

context ctx2 : boolean

observation o1 : ctx1>25 -> ctx2
```

o1 is satisfied every time the value of ctx2 changes to true after ctx1>25 was also updated to true. Note that the pattern is equivalent to:

eventWhen(ctx1>25) -> eventWhen(ctx2)

Pattern expressions: followed-by

```
observation o1 : ctx1 > 25 -> ctx2
```

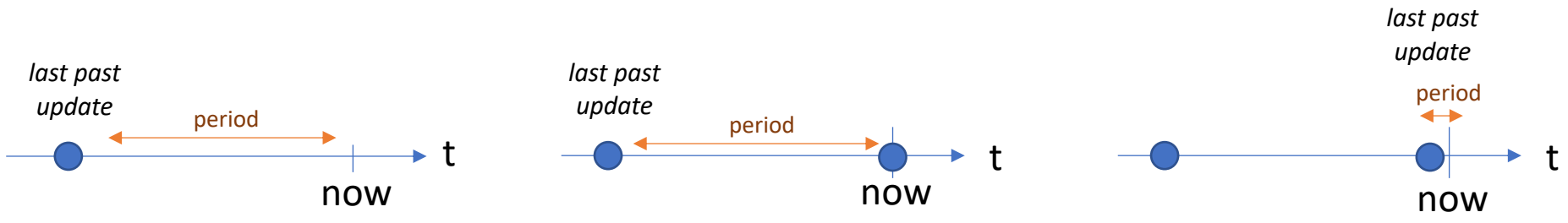


Pattern expressions: period

period(*variable*)

Arguments: the target variable

Output: the function "period" returns the period from the last update in the past



Example:

```
observation o1 : period(ctx1) < 20sec
```

It is satisfied if the period between two consecutive updates is less than 20 seconds

Pattern expressions: repeat

expr1 **repeat**(*n*)

Inputs: (1) An event-typed expression and (2) an integer

Output: produces an event when *expr1* is met *n* times

Examples:

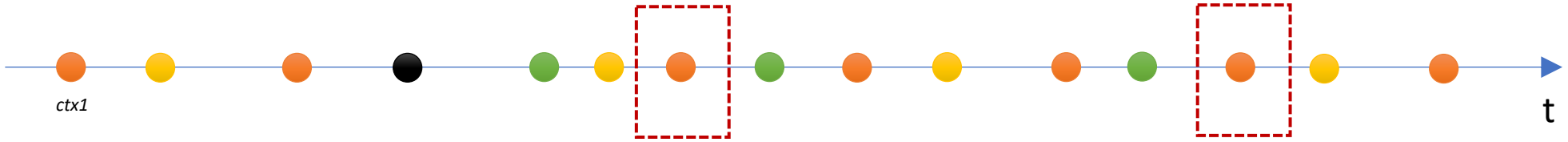
```
context ctx1 : number

observation o1 : ctx1 repeat(3)
```

o1 is satisfied every time ctx1 is updated 3 times. Note that it is equivalent to the following expression:

update(ctx1) repeat(3)

Pattern expressions: repeat

```
observation o1 : ctx1 repeat(3)
```



Pattern expressions: range

expr1 **range**(*n1*,*n2*)

Inputs: (1) An event-typed expression and (2)(3) integers

Output: produces an event when *expr1* is met between *n1* and *n2* times

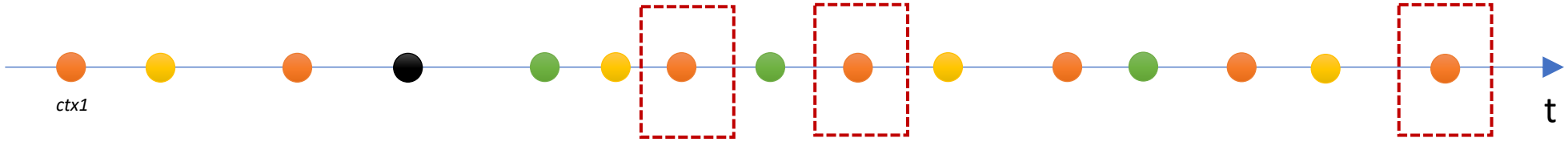
Examples:

```
context ctx1 : number

observation o1 : ctx1 range(3,4)
```

o1 is satisfied every time ctx1 is updated 3 times and 4, but not in the 5th update, in which case the counter is initialized again

Pattern expressions: range

```
observation o1 : ctx1 range(3,4)
```



Pattern expressions: once

once `expr`

Inputs: An event-typed expression

Output: produces an event only the first time *expr* is met. Note that, by the default, the pattern detection is executed in a loop, i.e., a pattern can be satisfied repeatedly

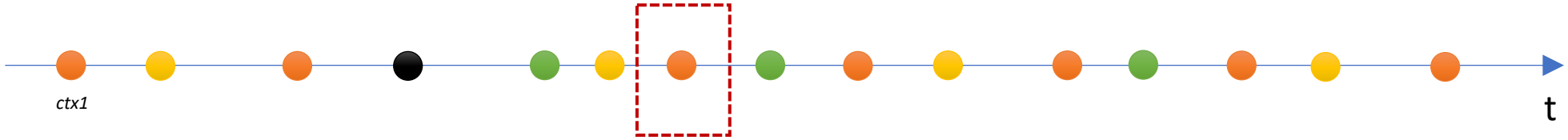
Examples:

```
context ctx1 : number

observation o1 : once ctx1 repeat(3)
```

o1 is satisfied after ctx1 receiving the 4rd update.
Without once, o1 would be satisfied every 3 updates.

Pattern expressions: once

```
observation o1 : once ctx1 repeat(3)
```



Pattern expressions: while

expr1 ***while***(*expr2*)

Inputs: (1) an event-typed expression, (2) a boolean expression

Output: produces an event every time *expr1* is met being *expr2* true

Examples:

```
context ctx1 : number

context state : enum {ST1, ST2, ST3}

observation o1 : ctx1>45 while(state=ST2)
```

o1 is satisfied when ctx1>45 changes to true while state is ST2

Pattern expressions: aggregation functions

*Variable'**aggregator**(args) or **aggregator**(variable, args)*

Arguments: (1) target variable, (2) value (window size, either: number of samples or time)

Output: value (numeric or Boolean)

Examples:

```
var aux1 : number := avg(ctx1, 30sec)
```

Temporal average of ctx1

Aggregation functions: *avg, min, max, count, sum, increasing, decreasing, stable*

Auxiliary elements

Importing RoQME models:

```
import "secondary.roqme"
observation o1 : mymodel.ctx1>45...
```

main.roqme

```
roqme mymodel
context ctx1 : number ...
```

secondary.roqme

Comments:

```
// This is a comment
```

Variable timer:

```
timer t1 := 37 min
observation o1 : t4 while(ctx>45)
```

Every 37 min checks if ctx > 45

Definition and use of data types:

```
type Qualifier : enum {LOW, MEDIUM, HIGH}
context battery : Qualifier
```

Auxiliary general-purpose variables:

```
var aux : number := 2
context ctx1 : number := aux * ctx2
```