ROBMOSYS-1FORC

RobMoSys

ROQME

DEALING WITH NON-FUNCTIONAL PROPERTIES THROUGH GLOBAL ROBOT QUALITY OF SERVICE METRICS

The RoQME Complex Event Processing Module: Mapping the RoQME modeling elements to EPL

(v1.0_201905022000)

THIS PROJECT HAS RECEIVED FUNDING FROM THE *EUROPEAN UNION'S HORIZON 2020 RESEARCH AND INNOVATION PROGRAMME* UNDER GRANT AGREEMENT NO. 732410, IN THE FORM OF FINANCIAL SUPPORT TO THIRD PARTIES OF THE ROBMOSYS PROJECT

Content

1 General Structure of the Generated EPL Module

We will describe the general structure of the EPL module generated from the observations included in the RoQME model.

```
[Imports]
      import org.roqme.cep.enums.*;
      uses model1;

[Event definitions]

      create schema DistanceEvent(val double);

[Variables, constantes and timers]

      create variable double max_distance = 100;

[EPL patterns]
      INSERT INTO Observations
      SELECT 'obs1'as obsName
      FROM pattern [ DistanceEvent(val>30) ] ;

[Last EPL pattern]
      @Name('Observations')
      SELECT obsName
      FROM Observations;
```

The first two blocks correspond to the import of classes (useful for importing data types defined outside the engine) and the definition of the events types which represents defined contexts. The following blocks would be the definition of variables, constants and timers.

In the last block we will define the complex event patterns. After defining all the EPL sentences that form the observations, we will end the block of patterns with a SELECT to the stream of Observations events, where the triggered observations will arrive.

With the *@Name* annotation, we will give a name to this last statement in order to be able to reference it within the Java code, and thus be able to assign it a listener.

2 Import Section

In this section we will find the import of enumerated classes, with the reserved keyword **import**, and the import of other EPL modules with the reserved keyword **uses**.

If we import a *.roqme* model, we will have to import the module corresponding to the imported model. This will allow us to make both models available to the engine.

2.1 Importing RoQME Models

When we import a *.roqme* model, in the generated EPL module we use the Esper reserved keyword **uses** to import the correspondent *.epl* module.

This work uses the *PatternDetector* API *launch* method. This method receives a number of modules (the principal one and all its dependencies) and returns them in the correct order to be deployed.

3 Mapping the RoQME Contexts

In the following examples we can see how to declare in Esper the event types corresponding to different context types.

```
create objectarray schema ContextA(val double, ts long) starttimestamp ts
endtimestamp ts;

create objectarray schema ContextB(val bool, ts long) starttimestamp ts endtimestamp
ts;

create objectarray schema ContextC(val EnumType, ts long) starttimestamp ts
endtimestamp ts;
```

We include 'starttimestamp *ts* endtimestamp *ts*' to let the event engine know that *ts* is the property that stores the timestamp.

3.1 Creating and Using Datatypes

In this case, we use the Esper inheritance feature for defining two contexts (*ContextA* and *ContextB*) that inherit from an event type *TypeA*.

```
create objectarray schema TypeA(val datatype, ts long) starttimestamp ts
endtimestamp ts;

create objectarray schema ContextA() starttimestamp ts endtimestamp ts inherits
TypeA;

create objectarray schema ContextB() starttimestamp ts endtimestamp ts inherits
TypeA;
```

4 Defining Constants and Variables

In order to map the RoQME parameters and variables into Esper we use the following equivalence:

- RoQME **variable** = Esper **variable**
- RoQME **param** = Esper **constant**

```
create variable variable_type variable_name = value;
```

```
create constant variable constant_type constant_name = value;
```

5 Defining Timers and Schedules

We define a map structure gathering all the timers included in the *.roqme* model. This map contains a sequence of pairs (*timer_name*, *timer_EPL_expression*). Then, when we find a *timer_name* in the specification of an observation included in the *.roqme* model, we replace its name with the corresponding *timer_EPL_expression*.

RoQME timers are mapped to Esper timers, declared as follows:

timer : interval

RoQME schedules are also mapped to Esper timers, in this case declared as:

timer : at

Consider the following excerpt of a *.roqme* model:

```
timer t1 := 15 “min”;
observation o1 : t1 and not ctx1 reinforces safety
```

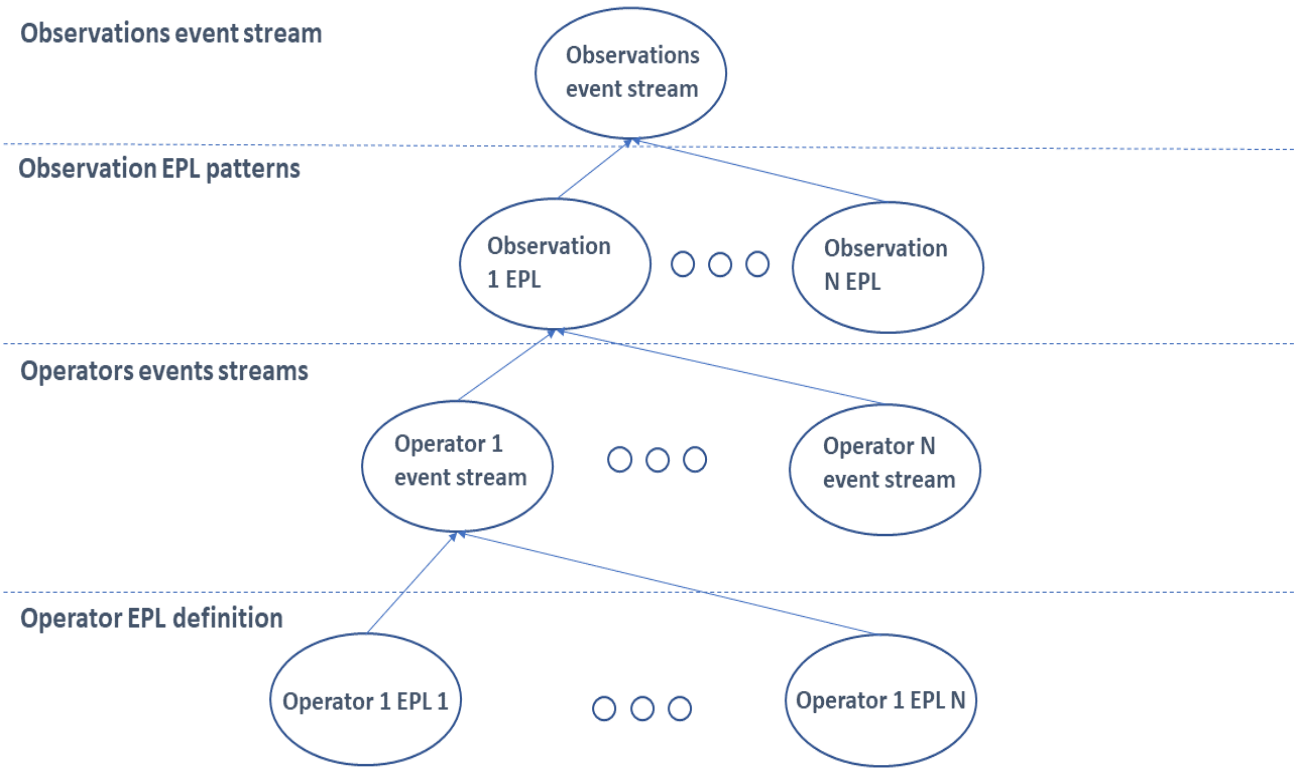
It will be translated into Esper as follows:

```
INSERT INTO Observations
SELECT 'o1' as val
FROM pattern [ every ( timer:interval(15 min) AND not ctx1 ) ];
```

6 Mapping RoQME Observations

RoQME observation may contain more than one operator which, in turn, may require more than one EPL statement.

In order to create relationships between the operators included in each RoQME observation without having to create such relationships between the corresponding EPL statements, several operators will have a dedicated stream of events.



Each RoQME operator may be mapped to one or more EPL statements. Each statement, when triggered, inserts a new event in the corresponding operator's event stream. Consider the following example:

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every ( EventStream(val=false) ->
                       EventStream(val=true) )];
```

The previous sentence inserts a new event in the event stream ***OPERATOR_STREAM_NAME*** when it detects a change from false to true in the ***EventStream***. As it can be noticed, we use generic names such as ***EventStream*** or ***OPERATOR_STREAM_NAME***, since these parts do not depend on EPL but on the RoQME model.

The name of the event streams associated to each RoQME operators is formed as follows:

OPERATOR_STREAM_NAME = {OBSERVATION_NAME | DERIVED_CONTEXT_NAME} + **OPERATOR_SHORT_NAME [** + **OPERATOR_PLACE** (If repeated)]

In order to create relationships between the streams associated to each operator we use Esper logical, relational, causal and temporal operators, creating a single EPL statement for each observation. When an

observation is fired, we create and insert a new event with its corresponding name into the event stream associated to that observation.

For example, consider the following observation:

```
observation Obs1 : eventWhen(ctx1>25) -> eventWhen(ctx1>50)
```

From the previous observation we will obtain, among others, an EPL sentence gathering the different operators it involves. This sentence will be responsible for inserting a new event in the event stream *Observations,* aimed at collecting all the triggered observations. Thus, for the previous observation we will obtain:

```
INSERT INTO Observations
SELECT 'Obs1' as val
FROM pattern [ every ( Obs1EW -> Obs1EW1) ];
```

To trigger the 'Obs1' observation, it is necessary to receive an event that comes from the first operator *eventWhen* (Obs1EW1), followed by an event coming from Obs1EW2.

6.1 EPL Patterns Associated to RoQME Operators

Due to their complexity or because we need to perform additional operations to achieve a certain functionality, there are some RoQME operators that need more EPL statements than others and, as a consequence, a dedicated event stream. This allow us to create relationships between operator event streams instead of using a huge unique EPL statement.

We need to define these auxiliary patterns before defining the observation pattern they belong to. Besides, if there are several nested operators, we must first generate the most internal ones.

To do this, we will go through the operator tree that contains the observation in the bidder, generating first the most internal operators. In this section we describe the different types of operators that require defining a dedicated event stream, based on the expressions package, included in the RoQME metamodel.

6.1.1 BinaryArithOp

For arithmetic operations, the value of the result must be updated whenever any of its operands changes. This is an obstacle because in CEP and Esper, by default, the information is not stored (it is only available the time necessary for processing it and then it is discarded).

In order to address this limitation, we request the event engine not to discard the last event for any of the operands involved in the arithmetic operation so that, if any operand receives an update, the result can be correctly recalculated using the last value associated to all the other operands.

For this we will use the window lastevent, which provides and stores the last event of the event stream where it is applied. This means that we will apply the window lastevent to all the operands.

Assuming that ***EventStreamA*** and ***EventStreamB*** are numeric contexts:

```
INSERT INTO ARITHMETIC_OPERATION_STREAM_NAME
SELECT a.val / b.val as val, current_timestamp as ts
FROM EventStreamA#lastevent as a, EventStreamB#lastevent as b;
```

6.1.2 NaryTermOp

The different situations related to *NaryTermOp* operations are described next, in the following subsections.

6.1.2.1 Aggregation Functions

In order to generate the necessary event patterns to implement aggregation functions, we will group them as follows:

1. Those aggregation functions that have an equivalent in Esper; and
2. Those that do not have an equivalent in Esper (namely: *stable*, *increasing* and *decreasing*) and therefore must be manually implemented.

Next, we include several examples of aggregation functions belonging to the first group, i.e, that have a direct correspondence with equivalent functions in Esper:

Average value for the last 30 seconds

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT avg(val) as val, current_timestamp as ts
FROM EventStreamName#time_batch(30 sec);
```

Average value for the last 30 events greater than 50

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT avg(val) as val, current_timestamp as ts
FROM EventStreamName#length_batch(30) as a
WHERE a.val > 50;
```

Summation for the last 3 events

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT sum(val) as val, current_timestamp as ts
FROM EventStreamName#length_batch(3);
```

Minimum value for the last 3 events

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT min(val) as val, current_timestamp as ts
FROM EventStreamName#length_batch(3);
```

Maximum value for the last 3 events

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT max(val) as val, current_timestamp as ts
FROM EventStreamName#length_batch(3);
```

Count of events that meet a certain condition for the last 100 events.

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT count(*) as val, current_timestamp as ts
FROM EventStreamName#length_batch(100)
WHERE expr;
```

In the case of the *increasing*, *decreasing* and *stable* aggregation functions, there is not a direct correspondence in Esper and, as a consequence, we have to manually implement them. Next you can find the mapping of a RoQME sentence including the *stable* function (highlighted in blue) into Esper:

```
stable(ctx1,20 "sec")
```

```
INSERT INTO INTO OPERATOR_STREAM_NAME
SELECT window(b.val=a.val).allOf(v=>v=true) as val, current_timestamp as ts
FROM pattern [ every ( a=ctx1 -> b=ctx1 ) ]#time_batch(20.0 sec);
```

We use the ***window*** directive with a logical expression (b.val=a.val) to retain the collection of the resulting comparisons. Then, we apply the Esper lambda function (*allOf*) to the previous collection so that, if all the resulting comparisons are true, it implies that the value keeps stable.

Note that the implementation of the *increasing* and *decreasing* aggregated functions is identical to the previous one, except for the logical operator used in the expression included inside the *window* directive (**=** for stable, **>** for increasing, and **<** for decreasing).

6.1.2.2 Arithmetic Functions

All the arithmetic functions available in RoQME have an equivalent in Esper. More specifically, Esper imports the java.Math class by default, enabling the use of all its arithmetic operations. Next, you can find a couple of examples of how RoQME expressions including arithmetic functions are mapped to Esper:

Numeric context example: `sqrt(EventStreamName)`

```
INSERT INTO INTO OPERATOR_STREAM_NAME
SELECT Math.sqrt(val) as val, current_timestamp as ts
FROM EventStreamName;
```

Numeric filtered context example: `sqrt(EventStreamName >20)`

```
INSERT INTO INTO OPERATOR_STREAM_NAME
SELECT Math.sqrt(val) as val, current_timestamp as ts
FROM EventStreamName(val>20);
```

6.1.2.3 Pattern Functions

eventWhen (expr)

The *eventWhen* operator is aimed at detecting when a condition is met, i.e., when the condition evaluation changes from false to true. Depending on the type of the expression passed as an argument (*expr*), the operator behaves differently:

- If *expr* is a Boolean variable, an event is triggered when, being that variable false, it is set to true.
- If *expr* is a logical expression, an event is triggered when, being the result of evaluating the expression false, it changes to true.

For Boolean variables…

The first statement, included below, uses the *firstevent* window. This window retains only the first event of the event stream. In the WHERE clause we verify that the value of the variable is true. This check is necessary since the following statement needs two events to trigger, and we have to check whether the first event meets the imposed conditions.

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM EventStreamName#firstevent as a
WHERE a.val = true;
```

The following statement is where the behavior of the operator *eventWhen* is defined. A new event will be inserted for the *eventWhen* operator only if we receive an event with value false followed by an event with value true.

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every (EventStream(val=false)->EventStream(val=true)) ];
```

For logical expressions…

In this case, the mapping is similar but replacing the true/false literals with the corresponding logical expression (*expr*):

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM EventStreamName#firstevent as a
WHERE a.val expr;

INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [every(EventStreamName(not expr)->EventStreamName(expr))];
```

For logical expressions depending on several contexts we generate the following variant:

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM EventStream#firstevent as a, EventStreamNameB#firstevent as b
WHERE a.val exprA Op b.val exprB;
```

IMPORTANT: The following variant indicates how to translate the *eventWhen* operator when the input logical expression appears negated (i.e., when it appears preceded by the *not* operator). Using the *not* operator with a stream of events means modeling the absence of events, which is not the desired semantics. To solve this problem, we must negate the expressions of the filters applied to the event flows, and also the relational operator between the operators.

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every (
      ( EventStreamNameA(not exprA) notOp EventStreamNameB(not exprB) )
      ->
      ( EventStreamNameA(exprA) Op EventStreamNameB(exprB) )
)];
```

update (expr)

The *update* operator will create a new event as long as the value of *expr* is updated to a new value. This operator will discard consecutive updates with the same value. Depending on the type of the expression passed as an argument (*expr*), the operator behaves differently:

- If *expr* is a Numeric or a Boolean variable, an event will be triggered whenever the value of the variable is updated.
- If *expr* is a logical expression, an event will be triggered both when the condition changes from false to true and also when it changes from true to false.

For Numeric and Boolean variables, the mapping is as follows:

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM EventStreamName#firstevent;

INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every a=EventStreamName ->
      ( not EventStreamName AND EventStreamName(val!=a.val) ) ];
```

For logical expressions, the mapping is as follows:

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM EventStreamName#firstevent as a
WHERE a.val expr;

INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every ( EventStreamName(expr) ->
      EventStreamName(not expr) ) ];

INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every ( EventStreamName(not expr) ->
      EventStreamName(expr) ) ];
```

For logical expressions depending on several contexts we generate the following variant:

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM EventStreamNameA#firstevent as a, EventStreamNameB#firstevent as b
WHERE a.val exprA Op b.val exprB;

INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every (
      ( EventStreamNameA(exprA) Op EventStreamNameB(exprB) ) ->
      ( EventStreamNameA(not exprA) notOp EventStreamNameB(not exprB))
)];

INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every (
      ( EventStreamNameA(not exprA) notOp EventStreamNameB(not exprB))->
      ( EventStreamNameA(exprA) Op EventStreamNameB(exprB) ) )];
```

period (variable)

The *period* operator returns the time elapsed between two consecutive updates of the indicated *variable,* and it is translated as follows:

`period ( ctx1 )` is translated as:

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT (b.ts - a.ts)/1000 as val, current_timestamp as ts
FROM pattern [ every a=EventStreamName ->
      ( not EventStreamName AND b=EventStreamName(val!=a.val)) ];
```

6.1.2.4 Conditional Term

`EventStreamName > 30 ? case_true_value : case_false_value` is translated as:

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT CASE WHEN a.ts > 30.0
       THEN case_true_value
       ELSE case_false_value
       END
       as val, current_timestamp as ts
FROM EventStreamName#lastevent as a;
```

6.1.3 RangeTerm

`ctx1 > 20 range(2,3)` is translated as:

```
INSERT INTO OPERATOR_STREAM_NAME
SELECT current_timestamp as ts
FROM pattern [ every ( EventStreamName (val > 20.0) ) ]
HAVING count(*) >= 2 and count(*) <= 3;
```

6.2 EPL Patterns Associated to RoQME Observations

Each RoQME observation is mapped to a single EPL statement gathering the relationships existing among its operators. At this level, we will use the logical, relational, causal and temporal operators that Esper / EPL provide to create the corresponding relationships between the operator's event streams.

6.3 Logical and Event Operators

`EventStreamName > 60 and EventStreamName < 120 )` is translated as:

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every (  EventStreamName(val > 60.0) AND
                        EventStreamName(val < 120.0) ) ];
```

`EventStreamName > 60 -> EventStreamName < 120 )` is translated as:

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every (  EventStreamName(val > 60.0) ->
                        EventStreamName(val < 120.0) ) ];
```

6.4 Relational Operators

These operators are used to compare either one variable with a constant or two variables.

`EventStreamName > 60` is translated as:

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( EventStreamName(val > 60.0) )];
```

`EventStreamNameA > EventStreamNameB` is translated using the *while* operator as:

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( ( a = EventStreamNameA AND
                         b  = EventStreamNameB ) while(a.val > b.val) )];
```

6.5 Unary Operators

This section describes the mapping of all the unary operators except *range* (see Section 6.1.3).

6.5.1 The NOT Operator

6.5.1.1 Boolean negation

```
!EventStreamName>60
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( EventStreamName(val <= 60.0) )];
```

```
!(EventStreamName>60 and EventStreamName<120)
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every (  EventStreamName(val <= 60.0) OR
                        EventStreamName(val >= 120.0) ) ];
```

```
!EventStreamName
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( EventStreamName(val = false)) ];
```

```
!EventStreamName::Literal
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( EventStreamName(val != EventStreamName.Literal) ) ];
```

6.5.1.2 Event negation

In this case we will use the *not* Esper operator to indicate the absence of events in the corresponding event flows. We recommend to review the Esper documentation to understand the semantics of this operator.

```
not ( EventStreamNameA  and EventStreamNameB  )
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( not (EventStreamNameA AND EventStreamNameB) ) ];
```

6.5.2 Repeat operator

For an event pattern to trigger after a certain number of repetitions, we must write the number of repetitions in brackets before the expression.

```
EventStreamNameA repeat(N)
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( [N] EventStreamNameA ) ];
```

```
EventStreamNameA AND EventStreamNameB repeat(N)
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( [N] (EventStreamNameA AND EventStreamNameB) ) ];
```

6.5.3 Once operator

```
EventStreamNameA AND EventStreamNameB
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ every ( EventStreamNameA AND EventStreamNameB ) ];
```

```
once EventStreamNameA AND EventStreamNameB
```

```
INSERT INTO Observations
SELECT 'obsName' as val
FROM pattern [ EventStreamNameA AND EventStreamNameB ];
```

7 Mapping Derived Context Patterns

Derived context patterns usually return a value and, as a consequence, they require some extra operations to calculate that value. Thus, they will require creating and inserting (in an event stream, named like the derived context) a new event with the result.

7.1 Numeric Type

When we define a context of a numeric type, the defined pattern must be an arithmetic expression or any operator that returns a numeric result. In the following example, the two RoQME instructions included next:

```
context ctx1 : number
context ctx2 : number := avg(ctx1>30)
```

are translated into:

```
INSERT INTO ctx2AVG
SELECT avg(val) as val, current_timestamp as ts
FROM ctx1(val > 30.0);

INSERT INTO ctx2
SELECT a.val as val, current_timestamp as ts
FROM pattern [ every ( a = ctx2AVG ) ];
```

7.2 Enum Type

There are no operations defined on the enumerated type. Thus, we can only use literals or an expression returning an enumerated value. In the following example, the two RoQME instructions included next:

```
context ctx1 : number
context ctx2 : enum{A,B,C} := ctx1>30 ? ctx2::A : ctx2::B
```

are translated into:

```
INSERT INTO ctx2
SELECT CASE WHEN a.ts > 30.0 THEN ctx2.A ELSE ctx2.B END as val,
           current_timestamp as ts
FROM ctx2#lastevent as a;
```

7.3 Event Type

In this case, we only need to register the timestamp when the event was created. In the following example, the two RoQME instructions included next:

```
context ctx1 : number
context ctx2 : eventtype := ctx1>30
```

are translated into:

```
INSERT INTO ctx2
SELECT current_timestamp as ts
FROM pattern [ every ( ctx1(val > 30.0) ) ];
```

7.4 Boolean Type

When we define a context of Boolean type, the defined pattern must be a logical expression or any operator that returns a Boolean result. Specifically we will create two EPL sentences:

1. In the first EPL statement we will record an event with a value of true when the defined expression is evaluated to true.
2. In the second we will record an event with a value of false when the defined expression is evaluated to false.

In the following example, the two RoQME instructions included next:

```
context ctx1 : number
context ctx2 : boolean := ctx1>30
```

are translated into:

```
INSERT INTO ctx2
SELECT true as val, current_timestamp as ts
FROM pattern [ every ( ctx1(val > 30.0) ) ];

INSERT INTO ctx2
SELECT false as val, current_timestamp as ts
FROM pattern [ every ( ctx1(val <= 30.0) ) ];
```

8 Setting variables

For the assignment of variables we have used the ***on*** clause. The processing is as usual: we first generate the auxiliary patterns and, in the most internal pattern, we will use the ***on*** clause where we express the relationships between its possible operators.

```
var varName : number := EventStreamName
on pattern [ every ( a = EventStreamName ) ]
set varName = a.val;
```

Thus, every time we receive an ***EventStreamName*** event, we will update the value of the variable to the event value.

9 Java Application for Managing and Executing the .epl Files

The structure of the Java application responsible for instantiating the event engine and executing the different generated .epl files is as follows:

```
public class AppName {
     public static void main(String[] args) throws    RoqmeDDSException,
                                                       InterruptedException {

          AppName exec = new AppName ( );
          PatternDetector pd = new PatternDetector ( );

          try {
               pd.launch ( "AppName.epl" );

               synchronized ( pd ) {
                    exec.wait ( );
               }

          } catch ( Exception e ){
               e.printStackTrace ( );
          } finally {
               pd.destroy ( );
          }

     }
}
```

In case we want to import another RoQME resource, we will need to generate the following code:

```
public class AppName {
     public static void main(String[] args) throws    RoqmeDDSException,
                                                       InterruptedException {

          AppName exec = new AppName ( );
          PatternDetector pd = new PatternDetector ( );
          List imports = new ArrayList<String> ( );
```

```
        try {
             imports.add ( "imported_model.epl" );
             imports.add ( "AppName.epl" );

             pd.launch ( imports );

             synchronized ( pd ) {
                  exec.wait ( );
             }

        } catch ( Exception e ) {
             e.printStackTrace ( );
        } finally {
             pd.destroy ( );
        }

    }
```

In order to use another RoQME model, we will need to create a list of EPL generated models, and pass them to the engine through the PatternDetector API.