(TIBCO BusinessEvents)

Event Pattern Language – A guidebook of sorts

Version	Author	Date	Notes
0.1	Ashwin Jayaprakash	11/4/2009	Initial version
0.2	Ashwin Jayaprakash	11/9/2009	Minor corrections
0.3	Ashwin Jayaprakash	11/19/2009	Grammar sample
0.4	Ashwin Jayaprakash	11/24/2009	Added EBNF and explained pattern clauses
0.5	Ashwin Jayaprakash	11/24/2009	Corrected syntax of examples
0.6	Ashwin Jayaprakash	12/11/2009	Confirmed shipment aggregation example works

What and Why

It is a feature in BusinessEvents.

It is an easy to use framework to solve some of the simpler and more commonly occurring problems in BE/Complex Event Processing such as:

- Patterns in event streams
- Correlation across event streams
- Temporal/Time based event sequence recognition
- Duplicate event suppression
- Store and Forward

It integrates neatly with the other features in BE. It complements the Rule processing and Query processing features.

What it is not

It is not a meant to do Process modeling nor replace State-machines nor perform event search/querying.

It is not a replacement for Rules or Queries or Database or Cache or Messaging.

It does not discover new patterns but recognizes predefined event patterns.

Quick introduction

This feature is composed of 2 parts:

- Pattern description language
- API and Catalog functions
  - Design-time
  - Run-time

The (language) event pattern is described in a form that looks like a combination of SQL and Regular Expressions.

Simple example

SEE GRAMMAR SECTION FOR FULL GRAMMAR

A typical example would look like this:

VERIFY SYNTAX

```
define pattern /OrderTracker
using /Order as order, /Fulfillment as fulfillment, /Shipment as shipment
with
        order.customerId
        and fulfillment.customerId
        and shipment.customerId
starts with order
then fulfillment
then shipment
```

This example demonstrates a simple way to correlate events across 3 different streams. The pattern listens to all 3 event streams (Order, Fulfillment and Shipment) and raises an alert if they occur in the order specified - Order, Fulfillment and then Shipment.

SEE PATTERN OUTPUT LISTENER/NOTIFICATION FUNCTIONS

Similarities and Differences (between Patterns, Rules and Queries)

Pattern	Rules & Statemachines	Continuous Queries
Specify and identify event arrival sequence / Temporal order		
Recognize patterns	Drive business logic	Continuous computation over one or more stream(s) of events
Correlate across streams	Join condition	Query join
Dynamic deployment		
Templatized patterns		
Complex patterns with sub-patterns	Nested states	
	HA and FT	
Like primitive Statemachines	Statemachine's State transitions offer rich and powerful syntax	
		Windowing constructs
		Incremental Aggregates, Sorting and Joins

Pattern language

Grammar

The language is quite straight forward. Conceptually it is a combination of SQL and Regular expressions. The language is more English-like that Regular expressions.

The pattern string is composed of 4 parts:

1. define pattern …
2. using …
3. with …
4. starts with … then …

REFER APPENDIX FOR FULL GRAMMAR

- **define pattern …**
    - Specifies a unique name for this pattern being defined
- **using …**
    - Specifies the event types to subscribe to in the pattern and an alias to identify each unique event type
    - Multiple event types are separated by “and”
- **with …**
    - Specifies the event property to be used for correlations/subscriptions
    - Each event type can have one such property defined
    - “and” is used to separate definitions
- **starts with …**
    - This is where the actual event sequence or pattern is described
    - The event aliases are used to indicate the absence / occurrence of the event in the sequence
    - “then” is used as the conjunction
    - There are various sub-clauses to describe the nature and duration of the event occurrences
    - A sub-pattern is indicated by wrapping the event sequence in brackets - “(” and “)”

Sample pattern:

```
Legend:
"|" denotes choice.
Text colored denotes keyword.
"\ " is the escape character.

define pattern /Patterns/PatternA | "/My Ontology/My Patterns/ PatternA"

using   /Ontology/EventA | /My Ontology/My\ Patterns/EventA as a
        and  /Ontology/EventB | /My Ontology/My\ Patterns/EventB as b

with a.id | a.id = "some string" | a.id = 10 | a.id = 10.0 | a.id = 0.1d
            | a.id = 333333L | a.id = 333333l | a.id = false | a.id = False | a.id = $param1
            | a.id = $date(2009, 12, 25) | a.id = $dateTime(2009, 12, 25, 9, 48, 37, 0)
            | a.id = $javaUtilDate
      and b.id

starts with a
then b
then any one (a, b)
then all (a, b)
then ( (a then b) )
then within 10 milliseconds | seconds | minutes | hours | days b
then repeat 10 to 20 times a
then repeat $intParam2 to $intParam3 times b
then after $longParam minutes
then all ( (a then b), b )
```

Explicit Temporal / Time based constructs

The pattern grammar implicitly describes a sequence of events. Therefore there is an implicit time component in each pattern.

In addition to these, there are 3 additional constructs that enforce a stricter time based restrictions on a sequence:

- Occurs Within
    - Ensures that all the events described inside the "within" clause occur within the time span specified. The timer starts as soon as the event preceding this sub-pattern arrives
    - As soon as all the events in the sub-pattern occur in the correct sequence, the pattern instance moves to the next step beyond this "within" clause

- Occurs During

    - Ensures that all the events described inside the "during" clause occur within the time span specified. The timer starts as soon as the event preceding this sub-pattern arrives
    - Even if all the events in the sub-pattern occur in the correct sequence before the timer expires, the pattern does not move to the next step until after the timer expiry

- Occurs After
    - This clause does not accept any event or sub-pattern. The timer starts as soon as the event preceding this sub-pattern arrives
    - This is used to model event sequences where there is no activity for certain fixed periods of time

Event subscription

To create a pattern definition you have to specify the event types you are interested in. You also have to specify a property to be used for subscription and to correlate with other events.

There are 3 ways to subscribe to an event:

1. Correlation on a property:

    Specify a property that will uniquely identify the event or related events. All the events being used in the pattern instance must have the same value for the correlation to work.

    Each pattern instance has an Id which is derived from this correlation property's value. If there are going to be multiple pattern instances simultaneously, then these values have to be unique per pattern instance.

    Example:

    i. order.customerId and shipment.customerId
        - Here, order and shipment events that share the same customerId will be correlated

    ii. stockQuote.symbol and tradeOrder.symbol
        - Here, stockQuote and tradeOrder events that share the same symbol will be correlated

2. Exact match:

    Specify a property and an exact value.

    Example:

    i. order.customerId = "123-ABC-456"
        - Here, only the order with the specified customerId will be processed

    ii. tradeOrder.symbol = "ABCD"
        - Here, only the tradeOrder with the specified symbol will be processed

3. Hybrid:

    When correlating events from multiple streams, some events can use correlation and others can use exact matches.

    Example:

    i. shipment.customerId and packagingPrepare.state = "California"
        - Here, a combination of correlation field and exact match is used to specify a pattern

Some restrictions on using exact matches:

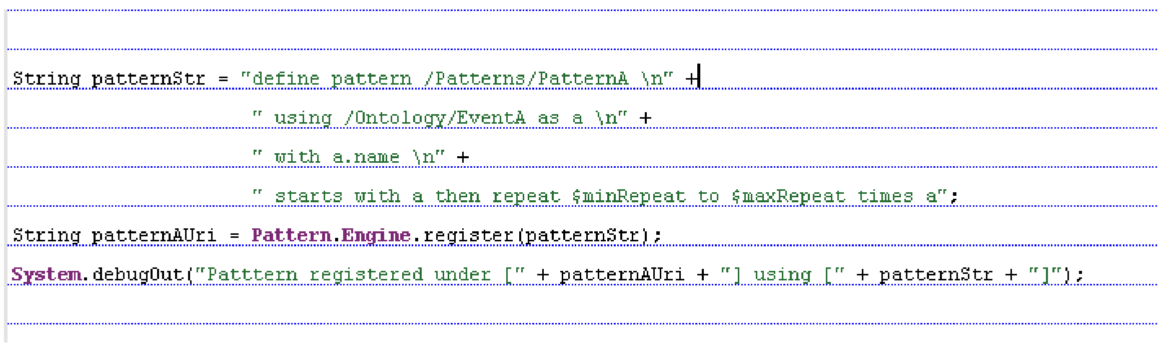
- There must be at least one event in the pattern that uses Correlation. As described earlier, the pattern instance's Id is derived from events' correlation property. Therefore at least the first event in the pattern sequence must use the correlation property
- If there is only one event in the pattern definition, then it should use Correlation
- If the first item in the pattern sequence is a "then-any-one" (*USE ACTUAL GRAMMAR REF*) or "then-all" (*USE ACTUAL GRAMMAR REF*), then all the events in that sub-sequence/item should use Correlation
- If an exact match is required for all events in the pattern instance then you are probably looking at the wrong place. Either a Rule or a Query will do the job. Or, convert all events to use Correlation instead

Details

LIST CATALOG FUNCTIONS

Pattern definition

Patterns have to be registered with the engine before deployment and use. Registration is a simple step which requires the full pattern string. The pattern name must be unique in the engine for the registration to succeed.



```
String patternStr = "define pattern /Patterns/PatternA \n" +
                    " using /Ontology/EventA as a \n" +
                    " with a.name \n" +
                    " starts with a then repeat $minRepeat to $maxRepeat times a";
String patternAUri = Pattern.Engine.register(patternStr);
System.debugOut("Patttern registered under [" + patternAUri + "] using [" + patternStr + "]");
```

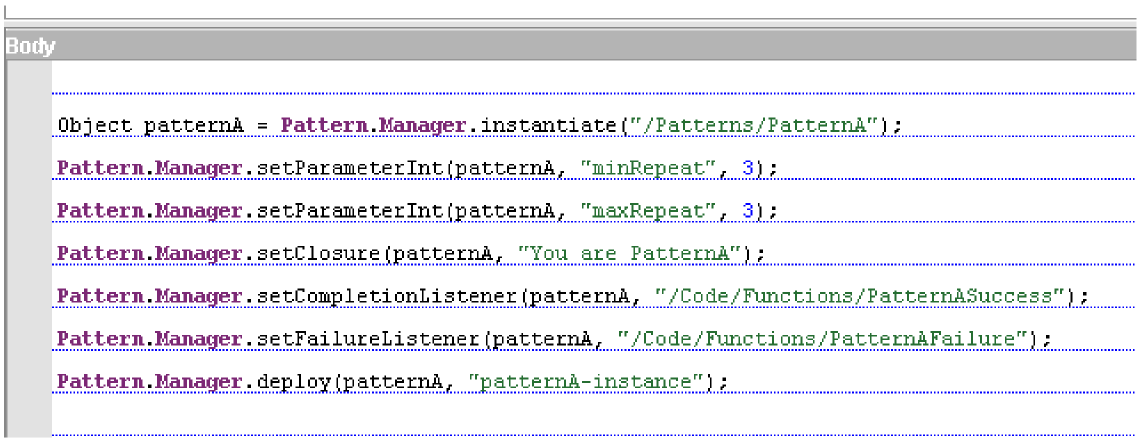
Pattern deployment

Patterns can be deployed and undeployed dynamically. The best time to do this would be at engine startup where the patterns can be instantiated, configured and deployed. Before shutting down the engine the patterns have to be undeployed.

Each RuleSession has a separate instance of the pattern framework. Unlike Statemachines or Rules running in certain modes, the patterns are neither distributed nor aware of the cluster. (*THIS MIGHT CHANGE IN A FUTURE VERSION*)

Before instantiating any pattern, the pattern framework should be started. This should typically be done in a Startup function. The framework also has to be shutdown when the engine shuts down.

SHOW CATALOG FUNCTIONS TO DO START, STOP, SETXXX, DEPLOY, UNDEPLOY

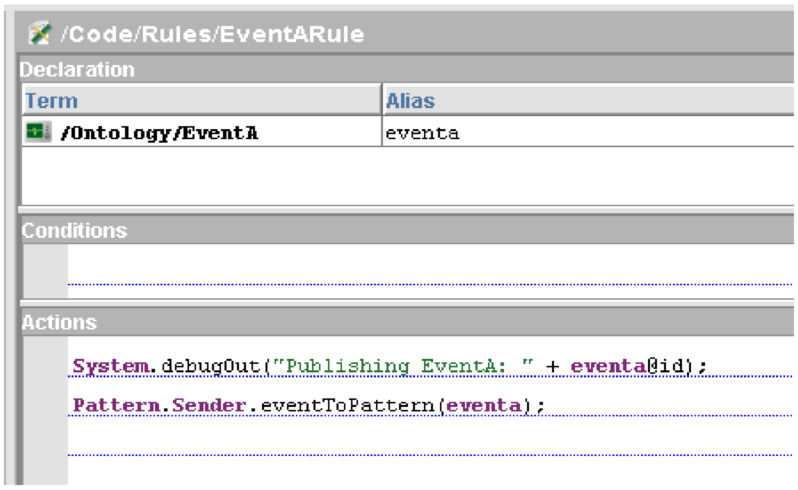


```
Object patternA = Pattern.Manager.instantiate("/Patterns/PatternA");
Pattern.Manager.setParameterInt(patternA, "minRepeat", 3);
Pattern.Manager.setParameterInt(patternA, "maxRepeat", 3);
Pattern.Manager.setClosure(patternA, "You are PatternA");
Pattern.Manager.setCompletionListener(patternA, "/Code/Functions/PatternASuccess");
Pattern.Manager.setFailureListener(patternA, "/Code/Functions/PatternAFailure");
Pattern.Manager.deploy(patternA, "patternA-instance");
```

Listening to events

Once a pattern is deployed, it is ready to start processing events. Events have to be explicitly sent to the pattern framework using the catalog function (*NAME THE CAT FN*). The pattern framework automatically routes these events to all subscribing patterns that have been deployed in the RuleSession.

This function can be invoked from anywhere in the context of a RuleSession. This call returns immediately because the actual work of routing to the pattern instances and the processing is done by other threads.



Term	Alias
/Ontology/EventA	eventa

```
System.debugOut("Publishing EventA: " + eventa@id);
Pattern.Sender.eventToPattern(eventa);
```

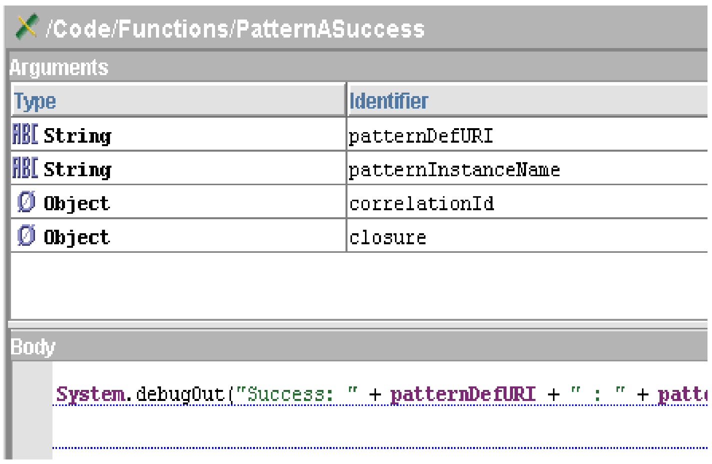
Pattern result listeners

Each pattern requires 2 listeners to be configured – a success listener and a failure listener. When a pattern instance successfully recognizes and completes the prescribed pattern it will invoke the success listener. If the pattern fails because of a timeout or a pattern arriving out of order, then it invokes the failure listener.

The listener can be of 2 types. A simple listener that offers basic notification features and another one that provides some insight into the events that triggered the pattern.

Time consuming operations must not be performed inside this listener. It should control return quickly to insure efficient functioning of the pattern framework.

Simple listener:



/Code/Functions/PatternASuccess

Arguments

Type	Identifier
String	patternDefURI
String	patternInstanceName
Object	correlationId
Object	closure

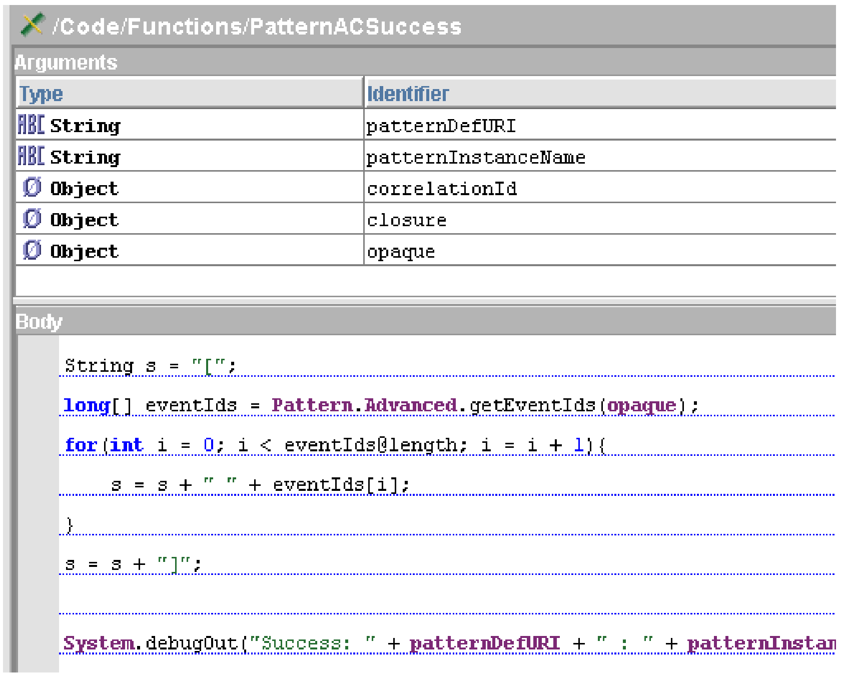
Body

```
System.debugOut("Success: " + patternDefURI + " : " + patt
```

Advanced listener:

Provides an additional parameter called “opaque”, which can be passed to functions that provide more details about the pattern instance.

SHOW GETEVENTIDS(..) AND GETRECENTEVENTID(..) FUNCTIONS



/Code/Functions/PatternACSuccess

Arguments

Type	Identifier
String	patternDefURI
String	patternInstanceName
Object	correlationId
Object	closure
Object	opaque

Body

```
String s = "[";
long[] eventIds = Pattern.Advanced.getEventIds(opaque);
for(int i = 0; i < eventIds@length; i = i + 1){
    s = s + " " + eventIds[i];
}
s = s + "]";

System.debugOut("Success: " + patternDefURI + " : " + patternInstan
```

Event references and pattern footprint

The reference (pointer) to an event that is sent to the pattern framework is discarded as soon as the pattern instances have processed it. This means that after the event properties have been extracted and the pattern instances have made their state transitions, the reference is discarded.

Once the property values are extracted, the pattern instances only retain the event Ids and not the events themselves.

This way apart from the internal data structures maintained by the pattern instances, there is no other memory overhead.

Other functions

TODESTINATION(..)

Events can created and sent to their default destinations using this function. The event gets sent immediately.

SETCLOSURE(..)

SETPARAMETERXX(..)

Examples

Simple correlation

Collect Order and Fulfillment events based on their Customer Ids.

```
define pattern /OrderTracker
using /Order as order, /Fulfillment as fulfillment
with order.customerId and fulfillment.customerId
starts with order then fulfillment
```

Simple temporal correlation

Collect Order and Fulfillment events based on their Customer Ids such that Fulfillment occurs within 10 minutes of placing the order.

```
define pattern /OrderFullfilmentSLA
using /Order as order, /Fulfillment as fulfillment
with order.customerId and fulfillment.customerId
starts with order  then within 10 minutes fulfillment
```

A pattern instance first gets created when the Order event arrives. If a corresponding Fulfillment event does not follow within 10 minutes of the Order event, then the Failure listener gets triggered. If the event does arrive on time, then the Success listener gets invoked.

Duplicate suppression / Store and Forward

Collect related events of the same type that share the same correlation id.

VERIFY SYNTAX
TEST THIS

```
define pattern /ShipmentAggregator
using /Shipment as shipment
with shipment.destinationState
starts with shipment
then within 2 hours ( starts with repeat 0 to 49 times shipment )
```

This pattern aggregates at most 50 Shipment events using the event's destinationState as the correlation property within a span of 2 hours.

When the first shipment event arrives, the pattern instance gets created. Then the timer starts and the pattern instance waits for 2 hours and accumulates a maximum of 49 more shipment events if they do arrive.

Not/Negation operator

There is no explicit operator for Negation/Not. The Not operator is meant to indicate a “Not occurs” situation.

However, most scenarios can be implemented by doing the following:

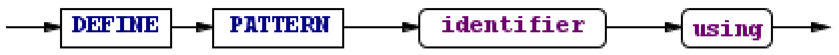
- Subscribe to the event type on which is not expected to occur
- Do not describe it in the pattern
- So, when the undesired event does occur due to the subscription, the pattern instance will fail

```
define pattern OrderFullfilment
using Order as order, Fulfillment as fulfillment, Cancellation as cancellation
where
        order.customerId
        and fulfillment.customerId
        and cancellation.customerId
starts with order
then within 10 minutes fulfillment
then after 5 minutes
```

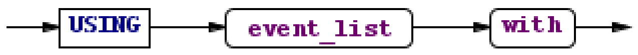
This pattern subscribes to Cancellation events but does not use it in the pattern. After the Order and the Fulfillment events arrive within the times specified, the pattern waits for another 5 minutes where it does not expect any input. If during this or any other time the Cancellation event occurs, then the pattern fails.

Elements in the Pattern language grammar:

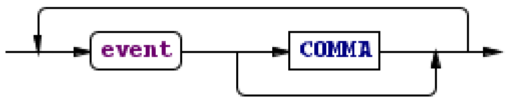
define_pattern



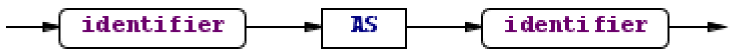
using



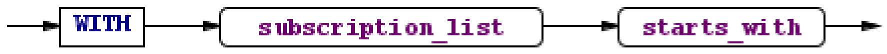
event_list



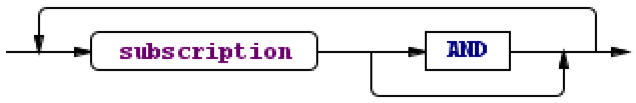
event



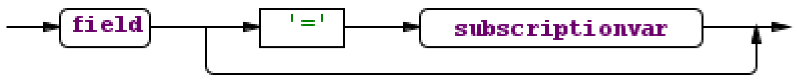
with



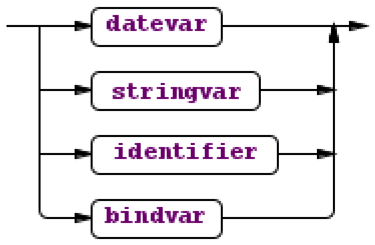
subscription_list



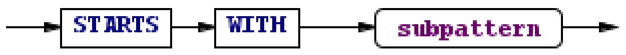
subscription



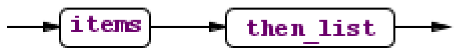
subscriptionvar

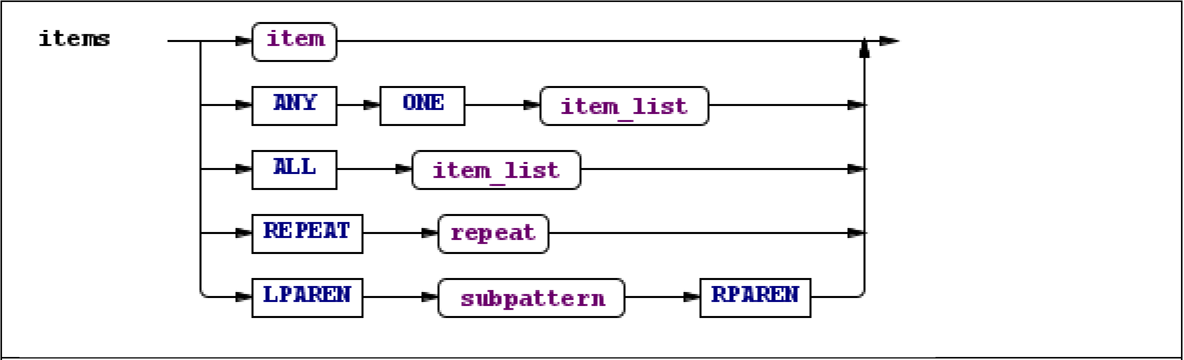


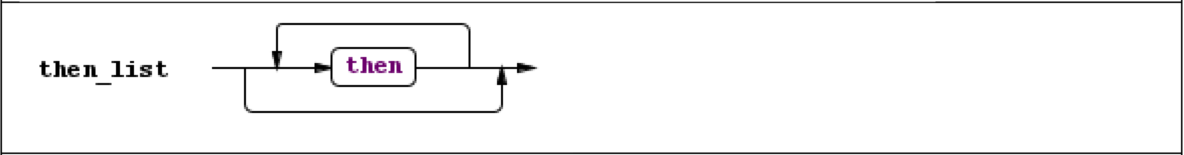
starts_with

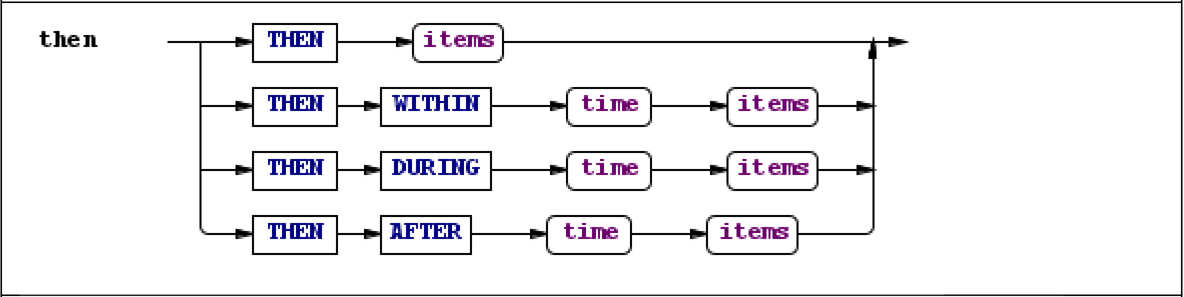


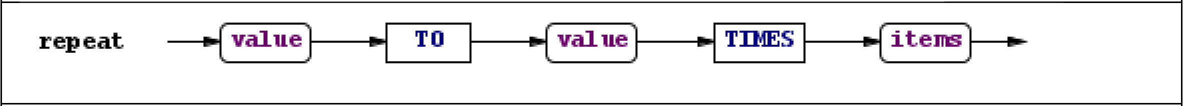
subpattern

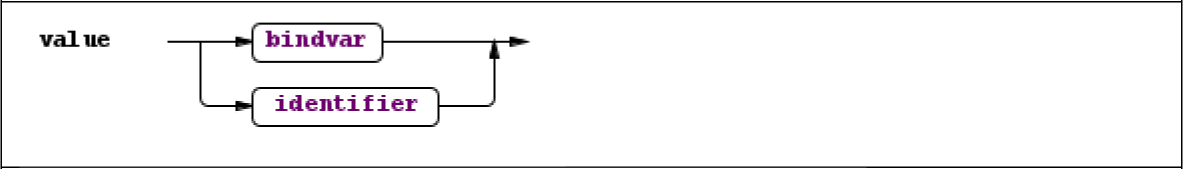


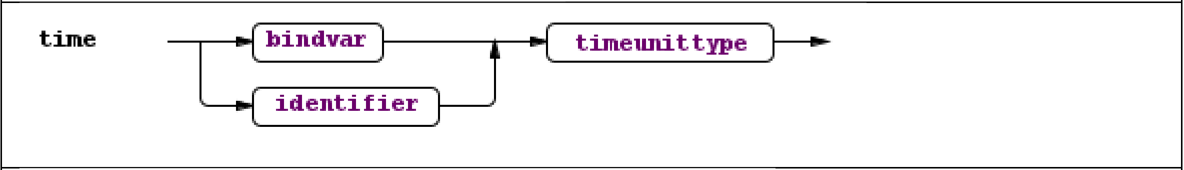

items
item
ANY
ONE
item_list
ALL
item_list
REPEAT
repeat
LPAREN
subpattern
RPAREN

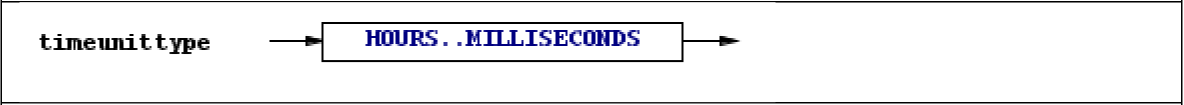

then_list
then

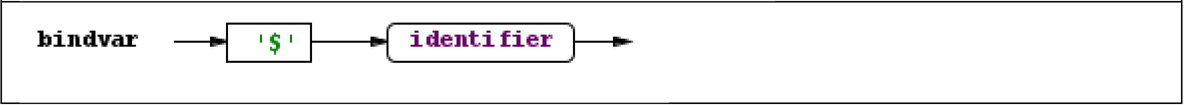

then
THEN
items
THEN
WITHIN
time
items
THEN
DURING
time
items
THEN
AFTER
time
items


repeat
value
TO
value
TIMES
items


value
bindvar
identifier


time
bindvar
identifier
timeunittype


timeunittype
HOURS..MILLISECONDS


bindvar
'$'
identifier

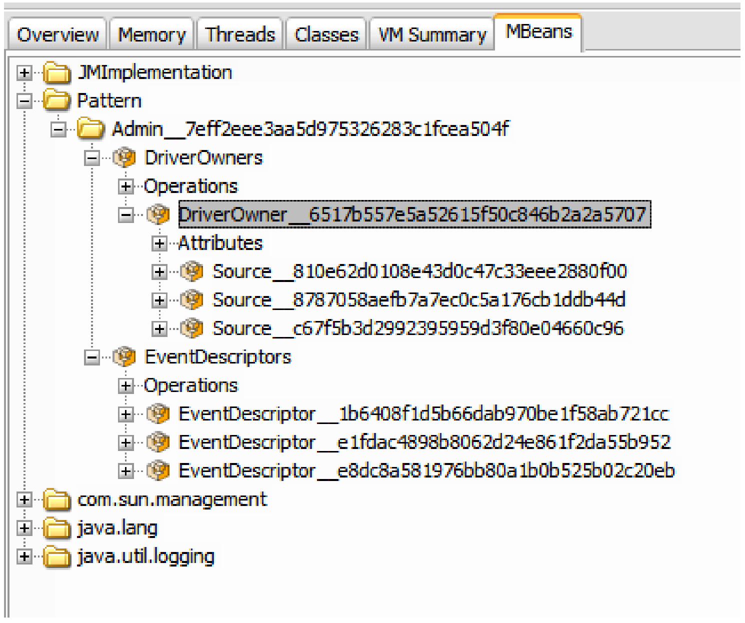
Others

JMX

The Pattern framework exposes a few JMX MBeans to list and view the various artifacts deployed in the engine.

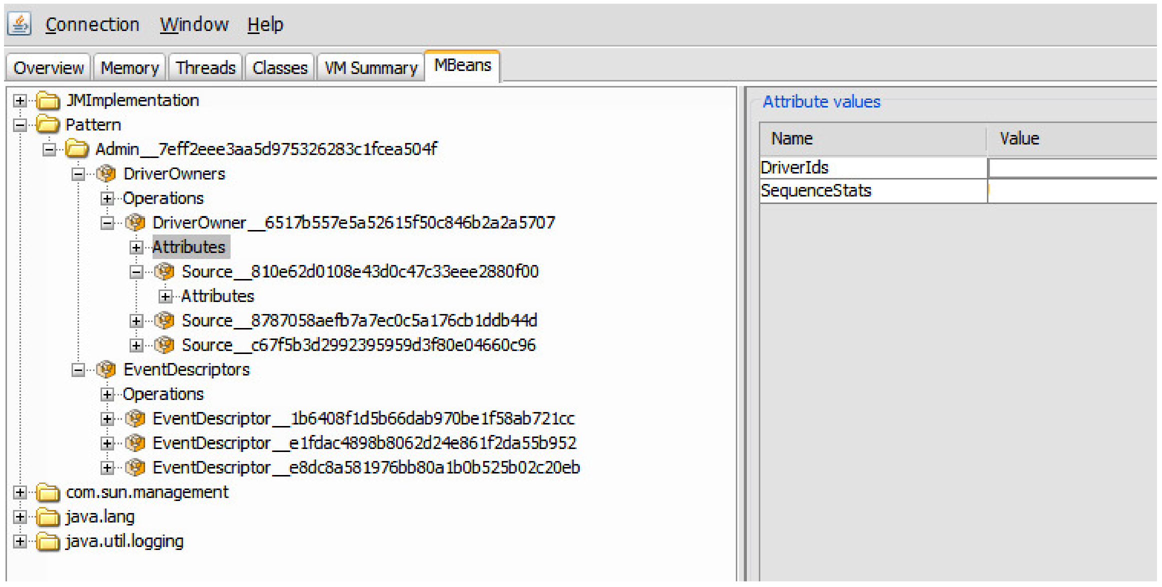
UPDATE THESE SCREENS

MBeans showing details about the various Driver Owners (pattern family) and the Event Descriptors (event sources):



```
JMImplementation
Pattern
  Admin__7eff2eee3aa5d975326283c1fcea504f
    DriverOwners
      Operations
      DriverOwner__6517b557e5a52615f50c846b2a2a5707
        Attributes
        Source__810e62d0108e43d0c47c33eee2880f00
        Source__8787058aefb7a7ec0c5a176cb1ddb44d
        Source__c67f5b3d2992395959d3f80e04660c96
    EventDescriptors
      Operations
      EventDescriptor__1b6408f1d5b66dab970be1f58ab721cc
      EventDescriptor__e1fdac4898b8062d24e861f2da55b952
      EventDescriptor__e8dc8a581976bb80a1b0b525b02c20eb
com.sun.management
java.lang
java.util.logging
```

Each Driver Owners also lists the ids of the Drivers (pattern instance ids):



```
JMImplementation
Pattern
  Admin__7eff2eee3aa5d975326283c1fcea504f
    DriverOwners
      Operations
      DriverOwner__6517b557e5a52615f50c846b2a2a5707
        Attributes
        Source__810e62d0108e43d0c47c33eee2880f00
          Attributes
        Source__8787058aefb7a7ec0c5a176cb1ddb44d
        Source__c67f5b3d2992395959d3f80e04660c96
    EventDescriptors
      Operations
      EventDescriptor__1b6408f1d5b66dab970be1f58ab721cc
      EventDescriptor__e1fdac4898b8062d24e861f2da55b952
      EventDescriptor__e8dc8a581976bb80a1b0b525b02c20eb
com.sun.management
java.lang
java.util.logging
```

Name	Value
DriverIds	
SequenceStats	

Limitations

- Each pattern instance is local to the engine
- No HA/FT (*As of 1.0*)
- Only Events are supported as sources
    - Concept data is derived from one or more Events, so Events are a natural fit to patterns

-- xxxx --