TIBCO BusinessEvents®
Event Stream Processing
Pattern Matcher Developer's Guide

Software Release 5.4
January 2017

Two-Second Advantage®

TIBCO®

Important Information

Contents

TIBCO Documentation and Support Services

Documentation for this and other TIBCO products is available on the TIBCO Documentation site. This site is updated more frequently than any documentation that might be included with the product. To ensure that you are accessing the latest available help topics, visit:

https://docs.tibco.com

Product-Specific Documentation

Documentation for TIBCO products is not bundled with the software. Instead, it is available on the TIBCO Documentation site. To directly access documentation for this product, double-click the following file:

TIBCO_HOME/release_notes/TIB_businessevents-eventstreamprocessing_*version*_docinfo.html where *TIBCO_HOME* is the top-level directory in which TIBCO products are installed. On Windows, the default *TIBCO_HOME* is `C:\tibco`. On UNIX systems, the default *TIBCO_HOME* is `/opt/tibco`.

The following documents for this product can be found in the TIBCO Documentation site:

- *TIBCO BusinessEvents Event Stream Processing Installation*
- *TIBCO BusinessEvents Event Stream Processing Pattern Matcher Developer's Guide*
- *TIBCO BusinessEvents Event Stream Processing Query Developer's Guide*
- *TIBCO BusinessEvents Event Stream Processing Release Notes*

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, contact TIBCO Support:

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

How to Join TIBCO Community

TIBCO Community is an online destination for TIBCO customers, partners, and resident experts. It is a place to share and access the collective experience of the TIBCO community. TIBCO Community offers forums, blogs, and access to a variety of resources. To register, go to the following web address:

https://community.tibco.com

Pattern Matcher Feature Overview

The Pattern Matcher add-on provide pattern-matching functionality, complementing TIBCO BusinessEvents rule processing and query processing features. Pattern Matcher consists of an easy-to-use language and a service that runs in a TIBCO BusinessEvents agent.

It addresses some of the simpler and more commonly occurring problems in complex event processing such as:

- Patterns in event streams
- Correlation across event streams
- Temporal (time based) event sequence recognition
- Duplicate event suppression
- Implementation of "Store and Forward" scenarios

Unlike rules or continuous queries, Pattern Matcher helps you to specify and identify the temporal order of event arrival.

The Pattern Matcher functionality can be used with any object management type. Its functionality is not dependent on or related to the object management layer.

The Pattern Matcher component listens to events that are explicitly sent to the service. It does not discover new patterns; given the patterns you define, it identifies those patterns in the event stream, returning valuable information you can make use of in your TIBCO BusinessEvents projects.

The pattern matcher service is not cluster aware. It operates within the scope of an agent. Keep this in mind when designing patterns. For example, in a multi-engine deployment, do not attempt to correlate events that may be received from a queue by different instances of an agent.

Components of Pattern Matcher

Pattern Matcher has two parts:

Pattern Description Language

The pattern description language is a straight-forward English-like language with similarities to SQL and regular expression languages. Within a simple syntax, however, you can specify complex patterns using nesting and various temporal constructs. You can also templatize patterns using bind variables. See Pattern Matcher Grammar for details.

Catalog Functions

Catalog functions for design time and deploy time enable you to dynamically deploy and undeploy patterns, specify values for bind variables, and specify success and failure listeners (callback functions) to take follow on action. You can also start and stop the pattern matcher service, though typically it is started and stopped when the TIBCO BusinessEvents engine itself starts and stops. See Pattern Matching Functions in a Project for details.

Comparison of Pattern Matcher and Other Components

Each component of Pattern Matcher has different use for different situation.

The following table shows some of the key features provided by Pattern Matcher, rules and state machines, and continuous queries. It enables you to decide which component to use for a given situation.

Comparison of Pattern Matcher and Other Components

Pattern Matcher	Rules, State Machines	Continuous Queries
Specify and identify event arrival sequence and temporal order		
Recognize patterns	Drive business logic	Continuous computation over one or more streams of events
Correlate across streams	Specify join conditions	Query join
Dynamic deployment		
Templatized patterns		
Complex patterns with sub-patterns	Nested states	
	High availability and fault tolerance	
Like primitive state machines	State transitions offer rich and powerful syntax	
		Windowing constructs
		Incremental aggregates, sorting, and joins

Pattern Matching Functions in a Project

Patterns can be deployed and undeployed dynamically in any agent that receives events such as a query agent or inference agent.

In a startup rule function you can start the service. You can also create, register, instantiate, configure, and deploy a pattern. In a shutdown rule function you can undeploy a pattern (which also unregisters the pattern). However you can perform these operations in different parts of your code, depending on the need.

See the online function reference in the HTML documentation for a list of functions.

Each agent has a separate instance of the pattern service. The patterns are not distributed. They are not cluster-aware.

Start the Pattern Matcher Service

In a startup rule function, start the pattern service, so it's running when the engine starts:

```
Pattern.Service.start();
```

You can start the Pattern Matcher service at any time after the processing unit starts. You can stop it any time before the processing unit stops. However it is generally advisable to start it in a startup rule function and stop the service in a shutdown rule function.

Send Events to the Service

You must send events to the service explicitly, for example in a rule that executes when the event is asserted. The actions (`then` block) would contain the following:

```
Pattern.IO.toPattern(MyEvent);
```

The pattern service automatically routes events of the specified type to all subscribing patterns that have been deployed in the agent.

You can invoke this call anywhere in the context of an agent, for example in a rule.

This call returns immediately because the actual work of routing to the pattern instances and the processing is done by other threads.

Create the Pattern String

The name of this String (`MyPatternString`) in the example) must be unique within the service. For example:

```
String MyPatternString = "define pattern /My/Pattern/URI \n" +
      " using /Ontology/EventA as a \n" +
      " and /Ontology/EventB as b \n" +
      " and /Ontology/EventC as c \n" +
      " with a.name and b.name and c.text = $ParamName \n" +
      " starts with a then b then c";
```

The value for bind variable `$ParamName` is provided in the `Pattern.Manager.SetParameterString()` function (see step).

Register the Pattern

Register the pattern string and get the pattern URI, which must be unique:

```
String MyPatternURI = Pattern.Manager.register(MyPatternString);
```

The pattern URI is any unique string. You can use slashes or dashes or simple text depending on how you want to organize the patterns using meaningful names. If the URI contains spaces, wrap the whole URI in double quotes (`"my name"`)

The pattern URI is also used to unregister the pattern.

Instantiate the Pattern Instance

Instantiate the pattern instance and set any bind variable values. You can instantiate a registered pattern multiple times, using a different instance name in each case and, as needed, different values for the pattern variables.

```
Object MyPatternInstance = Pattern.Manager.instantiate(MyPatternURI);
Pattern.Manager.setParameterString(MyPatternInstance, "ParamName", "ParamValue");
```

Set the Closure

The closure distinguishes one pattern instance from another:

```
      Pattern.Manager.setClosure(MyPatternInstance, "This is MyPatternInstance");
```

The closure for a pattern is used by listeners (callback functions), to distinguish one instantiated pattern instance from another.

Set the Listeners

Set the completion (success) listener and failure listener. See Success and Failure Listeners (Callback Functions) for more on these callback functions.

```
      Pattern.Manager.setCompletionListener(MyPatternInstance,
   "/RuleFunctions/MyPatternSuccess");
      Pattern.Manager.setFailureListener(MyPatternInstance,
   "/RuleFunctions/PatternSc2Failure");
```

Deploy the Pattern Instance

```
Pattern.Manager.deploy(MyPatternInstance, "DeployedPatternInstanceName");
```

The instance name is used to undeploy the instance.

Undeploy and Unregister a Pattern

Before shutting down the service undeploy and unregister the patterns.

```
Pattern.Manager.undeploy("MyPatternInstance");
Pattern.Manager.unregister("MyPatternURI");
```

Stop the Pattern Service

You can stop the Pattern Matcher service at any time before the processing unit stops (and you can also start it again) depending on need. It is generally advisable to stop the service in a shutdown rule function so that the service stops before the processing unit itself stops.

```
Pattern.Service.stop();
```

Success and Failure Listeners (Callback Functions)

For each instantiated and deployed pattern instance, you configure two callback rule functions. One acts as a success listener and the other as a failure listener.

Do not perform time consuming operations in a listener. A listener should return control quickly to insure efficient functioning of the pattern service.

The listeners execute every time a pattern succeeds or fails.

On successful completion of a pattern the service invokes the success listener. If the pattern fails because of a timeout or elements of the pattern arriving out of order, then it invokes the failure listener.

Success and failure listeners have the same arguments. Advanced listeners have an additional argument.

Functions that Cannot be Used in Listeners

Functions that read, modify, or delete concepts and events, such as `Instance.deleteInstance()`, cannot be used in the callback functions that you use as listeners. These functions must execute in the context of a run to completion cycle (RTC). They cannot be used in success or failure listeners, which run in a different thread. (See the functions documentation for details on thread pool management functions such as the `Pattern.Manager.Advanced.setPoolSize()` function.)

In order to use functions that execute in the context of an RTC, create a rule that executes the functions, and create an event with all the necessary information. Send the event using `Pattern.IO.toDestination()`, preferably on a local channel. The event is asserted in an RTC and triggers the rule, which executes the desired functions.

TIBCO BusinessEventsFunctions that are valid in the query engine are also valid in the pattern engine.

Function Documentation

For all function documentation, see the tooltips in the function catalog view in TIBCO BusinessEvents Studio. Tooltip text is also available in the online references available in the HTML documentation. Expand to CEP Pattern > Pattern in the function catalog.

Advanced Listeners

Advanced listeners (callback functions) can provide some insight into the events that triggered the pattern. Advanced listeners have an `opaque` object but otherwise the signature is the same as the simple listener.

The opaque object contains information about the events in the pattern set. Various provided functions enable you to get information from the object, for example, `Pattern.Advanced.getEventIds(`*`opaque`*`)`.

Listener Required Signature

Simple listeners (callback functions) must have a signature with the parameter types except the opaque parameter. The advanced listener also uses the opaque parameter.

See Pattern Matching Functions in a Project to understand how these parameter values are created.

Parameter	Notes
`String` *`patternDefURI`*	Identifies the URI of the registered pattern definition.

Parameter	Notes
`String` *`patternInstanceName`*	The name of the instantiated pattern instance. This name enables you to identify data belonging to different instantiations of the registered pattern definitions.
`Object` *`correlationId`*	This ID is derived from the first correlation property for each pattern set. The success or failure rule function is executed once for each set.
`Object` *`closure`*	Closure object provided when executing the Pattern.Manager.setClosure() function. The closure for a pattern is used to distinguish one instantiated pattern instance from another, generally used in completion and failure listeners.
`Object` *`opaque`*	Provides some insight into the events that triggered the pattern, using provided catalog functions.

Pattern Matching Examples

Some pattern matching example helps to understand pattern matching better.

Simple Correlation

Collects Order and Fulfillment events based on their Customer IDs.

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

Simple Temporal Correlation

Collects 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 and /Fulfillment as fulfillment
with order.customerId and fulfillment.customerId
starts with order then within 10 minutes fulfillment
```

A pattern instance is 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 is triggered. If the event does arrive on time, then the Success listener is invoked.

Duplicate Suppression – Store and Forward

Collects related events of the same type that share the same correlation ID.

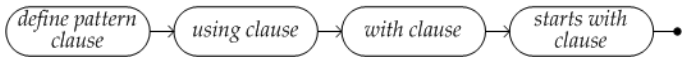
```
define pattern /ShipmentAggregator
using /Shipment as shipment
with shipment.destinationState
starts with shipment
then within 2 hours repeat 0 to 49 times shipment
```

This pattern aggregates at most 50 Shipment events within a span of two hours. it uses the event's `destinationState` as the correlation property. When the first shipment event arrives, the pattern instance is created. Then the timer starts and the pattern instance waits for two hours, accumulating a maximum of 49 more shipment events.

Pattern Matcher Grammar

A pattern string has four main clauses: *define pattern*, *using*, *with*, and *starts with* clause. These clauses define the patterns that need to be identified with specified streams.

The following figure displays the process flow of the four different clauses:



Syntax Example

The following is a simple example to illustrate the four clauses of a pattern. This example checks for an incorrect order of events in an order fulfillment and shipping flow.

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

The example demonstrates how Pattern Matcher correlates events across three different event streams. The pattern listens to all three streams (Order, Fulfillment and Shipment).

A pattern listens only to events that are sent to the Pattern Matcher service. See Send Events to the Service.

Syntax Diagrams

The syntax diagrams show the structure of a pattern and of each clause in a pattern.

Read the syntax diagrams from left to right. Items above or below a main line are optional. Items that can repeat are shown by lines that loop back from the end to the beginning of the repeating section, along with the separator character or word if one is required.

Miscellaneous Terms Used in Pattern Matcher Syntax Diagrams

alias	Alias for an event in the pattern.
identifier	A string that represents the name of a pattern or the URI of an entity. Identifiers
time unit	Allowable time units are as follows: `milliseconds`, `seconds`, `minutes`, `hours`, `days`

Names

- Pattern names (URIs) and event name can be any character inside double quotes, except double quote itself. Pattern URI needs quotes only if there is a space in the URI.
- Field and property names must be valid Java identifiers.
- Each alias must be globally unique in the whole pattern.
- To escape a keyword, use the pound sign (`#` – also known as a hash sign) before the keyword, for example, `#define`.
- Alias name, field name, property name, subscription field and bind variables can be use any of the following:

- alphanumeric
- digit
- underscore ('_')
- slash ('/')
- Escaped keywords.

Bind Variables

Variables begin with the dollar sign (`$`). The value is provided at deploy time. See Create the Pattern String.

You cannot use bind variables with the `datetime()` or `date()` functions. For example, it is not possible to use this type of call:

```
$datetime($year,$day,$month,....)
```

You can, however, use bind variables using the following function:

```
Pattern.Manager.setParameterDateTime()
```

For example:

```
Pattern.Manager.setParameterDateTime(patternSc14, "javaUtilDate", date);
```

Not (Negation) Scenarios

Although the language has no explicit operator for Negation or Not, negative scenarios can be implemented in other ways such as:

- Subscribe to the event type on which is not expected to occur.
- Do not describe it in the pattern.

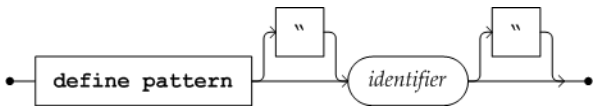
So, when the undesired event does occur due to the subscription, the pattern instance will fail. The following pattern example demonstrates a negation scenario:

```
define pattern OrderFullfilment
using Order as order and Fulfillment as fulfillment and 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 them in the pattern. After the Order and the Fulfillment events arrive within the times specified, the pattern waits for another five minutes, during which it does not expect any input. If during this or any other time the Cancellation event occurs, then the pattern fails.

Define Pattern Clause

The Define Pattern clause specifies a unique URI for the pattern.



The URI is used as a parameter in various catalog functions provided with the component.

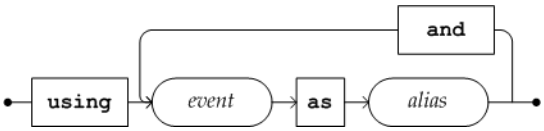
Examples

You can use any format that is useful to you in helping to identify the pattern. If you use spaces, use double-quotation marks around the URI:

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

Using Clause

The `using` clause specifies one or more event types to subscribe to in the pattern, and an alias for each. One says that the pattern subscribes to these events.



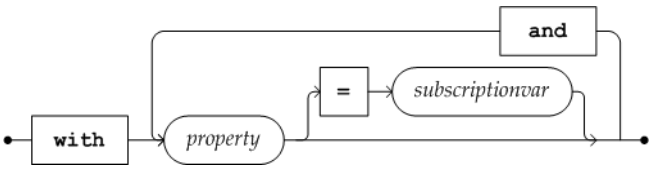
Use `and` to separate each event type.

Examples

```
using /Ontology/EventA as a and /Ontology/EventB as b
using "/My Ontology/My Patterns/EventA" as a
   and "/My Ontology/My Patterns/EventB" as b
```

With Clause

In the `with` clause of a pattern, you specify a property for each event in the using clause. You cannot use the event payload. Each event type can have one such property defined. The property or properties are used for correlations and subscriptions.



Optionally you can specify an exact match with a property value on the second or subsequent events listed in this clause. The specified property value must not match the value of any other property value in the pattern. Use `and` to separate each property.

The match is successful only if each event instance that arrives in the Pattern Matcher service occurs in the order specified in the `starts with` clause.

The first term in the `with` clause must use correlation. For details on this topic, see With Clause and the Correlation Property for details.

Examples

```
with a.id and b.id
with a.id and b.id = "some string"
```

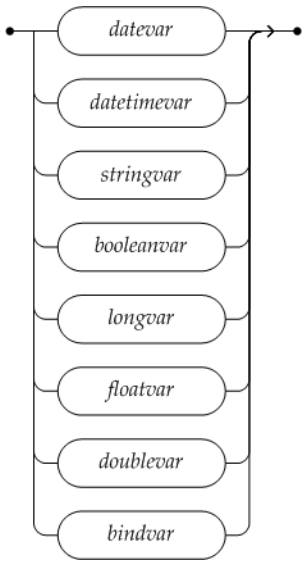
Correlation or Subscription Value

The *`subscriptionvar`* item specifies the correlation or subscription value.

The value can be one of the following:

- `datevar`
- `datetimevar`
- `stringvar`

- booleanvar
- longvar
- floatvar
- doublevar
- bindvar



The syntax for `datevar` and `datetimevar` are as follows:

```
$date(year,month,day,[,GMT offset])
$datetime(year,month,day,hour,minute,second,millisecond[,GMT offset])
```

Note that the `$date()` and `$datetime()` functions have an optional GMT timezone offset. This timezone offset is expressed as a plus sign (+) or a negative sign (-) followed by four digits, within double quotes. For example, to specify GMT minus eight hours and 30 minutes, use `"-0830"` As another example, use `"+0200"` to specify GMT plus 2 hours.

With Clause and the Correlation Property

The first term in a `with` clause must be a *correlation* property. It is also sometimes referred to as the *subscription* property.

Specify a property that will uniquely identify the event or related events. Instances of all the event types in the pattern that are being correlated must have the same property value for the correlation to succeed. For example if the correlation is `a.name and b.name` then the correlation succeeds if the value of `name` is `Joe` in both cases.

Fields that can have null values cannot be used as correlation properties. Such fields are ignored.

Each instance of a pattern has an ID which is derived from this correlation property's value. If multiple pattern instances exist simultaneously, the property values must be unique per pattern instance.

For example:

```
with order.customerId and shipment.customerId
```

In the above example, `order` and `shipment` events that share the same `customerId` will be correlated.

Here is an example of simple correlation:

```
with a.id and b.id
```

In the above case, the pattern succeeds when the following occurs:

- The value of the `id` property in an instance of event type `b` matches the value of the `id` property in an instance of event type `a`.
- And the order in which these two events arrives matches the order specified in the `starts with... then...` clauses.

You can use correlation with one event type. In this case the `starts with... then...` clauses specify the temporal conditions that instances of that event must meet.

Correlation and Exact Match

You can use one or more exact matches in your pattern. To use an exact match, specify a property and an exact value for that property.

In the following example, only the `order` event with the specified `customerId` property value will be processed.

```
order.customerId = "123-ABC-456"
```

A pattern cannot begin with an exact match; a correlation is required as the first element in the clause.

If the first item in the `then` subclause of the `starts with` clause is one of the following:

```
then any one
then all
```

Then all events in that sequence must use correlation.

Example

To illustrate how exact matches are used, consider the following simple example. Orders are placed and processed for shipment, then they sit on the loading bay, waiting for the next delivery truck. If the truck does not pick up the orders within two hours, customer service is alerted.

```
define pattern /OrderShipper
using /Order as order and /orderProcessed as processed and /deliveryvan as van
with
    order.customerId
    and processed.customerId
    and van.pickupStatus=Ready
starts with order
then processed
then within 2 hours van
```

The pattern is deployed and the Pattern Matcher service starts listening to the events that are sent to it. It puts sets of events that satisfy all the aspects of the pattern into "buckets." At a certain point in time, it has the following "buckets":

- `order.customerId=123`, `processed.customerId=123`
- `order.customerId=456`, `processed.customerId=456`
- `order.customerId=789`

Then a truck arrives and the loading bay staff enters its status. A `deliveryvan` event is sent with `status=ready`.

The Pattern Matcher updates all of its "buckets" for this pattern accordingly:

- `order.customerId=123`, `processed.customerId=123`, `van.pickupStatus=ready`
- `order.customerId=456`, `processed.customerId=456`, `van.pickupStatus=ready`

The above pattern instances succeed and the success rule function executes for each of the instances.

The bucket that contained only `order.customerId=789` fails, and the failure rule function pattern executes.

You might feel that the incomplete "bucket" with `order.customerId=789` should just wait for its corresponding event, `processed.customerId=789` and be delivered on a later truck. If that is the case

you must write a different pattern. This simple example only illustrates how the Pattern Matcher service processes a pattern that contains a correlation and an exact match.

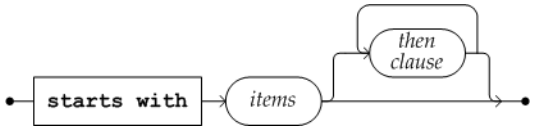
You can specify an exact match in many ways, such as the following:

```
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
```

Parameter values are provided at deploy-time.

Starts With Clause

A pattern describes a sequence of events, beginning with the `starts with` event, followed by each `then` event, in the order specified. The `starts with` clause is where the actual event sequence or pattern is described. For the pattern to succeed, all the events must be received according to the specified order (and any additional time constraints).



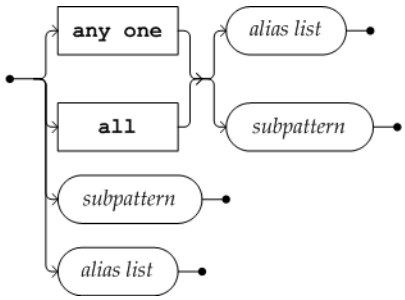
Only events listed in the `using` clause can be used. You specify the event sequence or pattern using the event aliases, to indicate the absence or occurrence of events in the sequence. Use `then` as the conjunction, for example:

```
starts with a then b then c
```

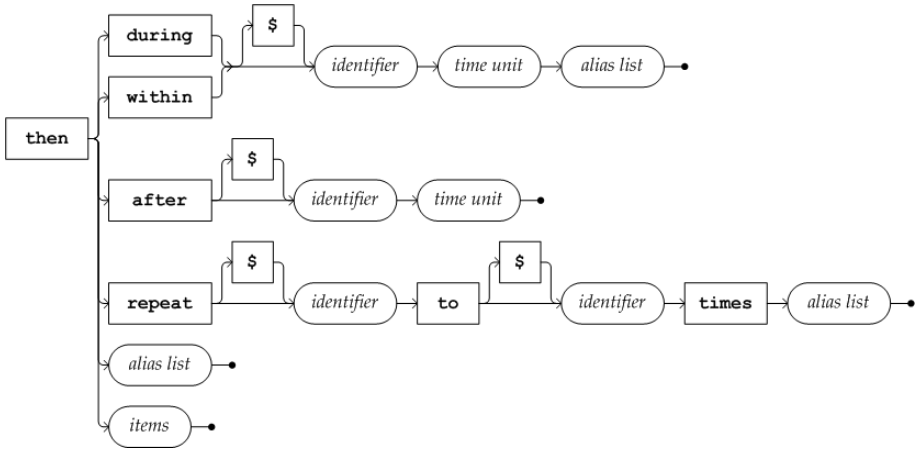
In the simplest case, the order in which the events must occur is signified by the order in which they are specified in the clause. However you can specify the ordering in a variety of ways. You can create sub-clauses and sub-patterns to describe constraints such as the number of event occurrences and the interval between them. To indicate a sub-pattern, wrap the event sequence in parentheses: `(`*`sub-pattern`*`)`.

Bind variables begin with the dollar sign (`$`). The value is provided at deploy time. See Create the Pattern String.

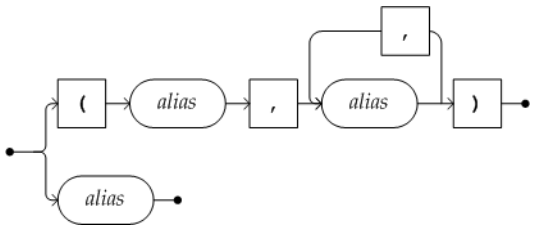
Items Syntax



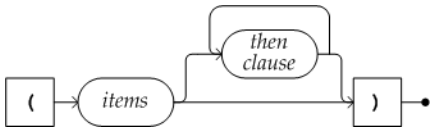
Then Syntax



Alias List Syntax



Sub-pattern Syntax



Starts With Sub-clause Examples

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

Then Sub-clause Examples

```
then any one (a, b)
then all (a, b)
then within 10 milliseconds
then repeat 10 to 20 times a
then repeat $intParam2 to $intParam3 times b
then after $longParam minutes
then all ( (a then b), b )
```

As shown in the example, a nested sub-pattern is surrounded by parentheses.

Clauses for Explicit Temporal Constructs

The pattern grammar implicitly describes a sequence of events: there is an implicit time component in each pattern. In addition three constructs enforce explicit time-based restrictions on a sequence: `within`, `during`, and `after`. You can use these constructs to enforce stricter time-based constraints on a sequence:

Within

The `within` construct 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 after the within clause.

During

Like `within`, the `during` construct 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.

The pattern remains in the `during` sub-pattern until the timer has expired, even if all the events in the sub-pattern occur in the correct sequence before that time. (This behavior is the difference between `occurs within` and `occurs during`.)

After

The `after` construct simply specifies a time period to wait before accepting the next event. It does not accept any event or sub-pattern. The timer starts as soon as the event preceding this sub-pattern arrives.

Use this construct to model event sequences where there is no activity for certain fixed periods of time.