JBoss Enterprise BRMS Platform 5

BRMS Complex Event Processing Guide

For JBoss Administrators

Edition 5.3.1

Last Updated: 2017-11-17

JBoss Enterprise BRMS Platform 5 BRMS Complex Event Processing Guide

For JBoss Administrators
Edition 5.3.1

Red Hat Content Services

Abstract

This guide is for developers and rule authors using JBoss BRMS Complex Event Processing with JBoss Enterprise BRMS Platform 5.2.0 and 5.3.1.

Table of Contents

PREFACE

CHAPTER 1. INTRODUCTION

1.1. INTRODUCTION TO COMPLEX EVENT PROCESSING

JBoss BRMS Complex Event Processing provides the JBoss Enterprise BRMS Platform with complex event processing capabilities.

For the purpose of this guide, *Complex Event Processing*, or CEP, refers to the ability to process multiple events and detect interesting events from within a collection of events, uncover relationships that exist between events, and infer new data from the events and their relationships.

An *event* can best be described as a record of a significant change of state in the application domain. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or even hierarchies of correlated events. Using a stock broker application as an example, a change in security prices, a change in ownership from seller to buyer, or a change in an account holder's balance are all considered to be events as a change has occurred in the state of the application domain.

Event processing use cases, in general, share several requirements and goals with *business rules use cases*.

From a business perspective, business rule definitions are often defined based on the occurrence of scenarios triggered by events. For example:

- On an algorithmic trading application: Take an action if the security price increases X% above the day's opening price.

  The price increases are denoted by events on a stock trade application.

- On a monitoring application: Take an action if the temperature in the server room increases X degrees in Y minutes.

  The sensor readings are denoted by events.

Both business rules and event processing queries change frequently and require an immediate response for the business to adapt to new market conditions, regulations, and corporate policies.

From a technical perspective:

- Both business rules and event processing require seamless integration with the enterprise infrastructure and applications. This is particularly important with regard to life-cycle management, auditing, and security.

- Both business rules and event processing have functional requirements like *pattern matching* and non-functional requirements like response time limits and query/rule explanations.

NOTE

JBoss BRMS Complex Event Processing provides the complex event processing capabilities of JBoss Business Rules Management System. The Business Rules Management and Business Process Management capabilities are provided by other modules.

Complex event processing scenarios share these distinguishing characteristics:

- They usually process large numbers of events, but only a small percentage of the events are of interest.
- The events are usually immutable, as they represent a record of change in state.
- The rules and queries run against events and must react to detected event patterns.
- There are usually strong temporal relationships between related events.
- Individual events are not important. The system is concerned with patterns of related events and the relationships between them.
- It is often necessary to perform composition and aggregation of events.

As such, JBoss BRMS Complex Event Processing supports the following behaviors:

- Support events, with their proper semantics, as *first class citizens*.
- Allow detection, correlation, aggregation, and composition of events.
- Support processing streams of events.
- Support temporal constraints in order to model the temporal relationships between events.
- Support *sliding windows* of interesting events.
- Support a *session-scoped* unified clock.
- Support the required volumes of events for complex event processing use cases.
- Support reactive rules.
- Support adapters for event input into the engine (pipeline).

The rest of this guide describes each of the features that JBoss BRMS Complex Event Processing provides.

Report a bug

CHAPTER 2. FEATURES OF JBOSS BRMS COMPLEX EVENT PROCESSING

2.1. EVENTS

Events are a record of significant change of state in the application domain. From a complex event processing perspective, an event is a special type of fact or object. A fact is a known piece of data. For instance, a fact could be a stock's opening price. A rule is a definition of how to react to the data. For instance, if a stock price reaches $X, sell the stock.

The defining characteristics of events are the following:

Events are *immutable*

An event is a record of change which has occurred at some time in the past, and as such it cannot be changed.

NOTE

The rules engine does not enforce immutability on the Java objects representing events; this makes *event data enrichment* possible.

The application should be able to populate un-populated event attributes, which can be used to enrich the event with inferred data; however, event attributes that have already been populated should not be changed.

Events have strong *temporal constraints*

Rules involving events usually require the correlation of multiple events that occur at different points in time relative to each other.

Events have *managed life-cycles*

Because events are immutable and have temporal constraints, they are usually only of interest for a specified period of time. This means the engine can automatically manage the life-cycle of events.

Events can be declared as either *interval-based* events or *point-in-time* events. Interval-based events have a duration time and persist in working memory until their duration time has lapsed. Point-in-time events have no duration and can be thought of as interval-based events with a duration of zero.

Report a bug

2.2. EVENT DECLARATION

To declare a fact type as an event, assign the `@role` meta-data tag to the fact with the `event` parameter. The @role meta-data tag can accept two possible values:

- `fact`: Assigning the fact role declares the type is to be handled as a regular fact. Fact is the default role.
- `event`: Assigning the event role declares the type is to be handled as an event.

This example declares that a stock broker application's **`StockTick`** fact type will be handled as an event:

Example 2.1. Declaring a Fact Type as an Event

```
import some.package.StockTick

declare StockTick
    @role( event )
end
```

Facts can also be declared inline. If **`StockTick`** was a fact type declared in the DRL instead of in a pre-existing class, the code would be as follows:

Example 2.2. Declaring a Fact Type and Assigning it to an Event Role

```
declare StockTick
    @role( event )

    datetime : java.util.Date
    symbol : String
    price : double
end
```

For more information on *type declarations*, please refer to the Rule Language section of the *JBoss Rules Reference Guide*.

Report a bug

2.3. EVENT META-DATA

Every event has associated meta-data. Typically, the meta-data is automatically added as each event is inserted into working memory. The meta-data defaults can be changed on an event-type basis using the meta-data tags:

- @role
- @timestamp
- @duration
- @expires

The following examples assume the application domain model includes the following class:

Example 2.3. The VoiceCall Fact Class

```
/**
 * A class that represents a voice call in
 * a Telecom domain model
 */
public class VoiceCall {
    private String  originNumber;
    private String  destinationNumber;
    private Date    callDateTime;
    private long    callDuration;          // in milliseconds

    // constructors, getters, and setters
}
```

@role

The @role meta-data tag indicates whether a given fact type is either a regular fact or an event. It accepts either **`fact`** or **`event`** as a parameter. The default is **`fact`**.

```
@role( <fact|event> )
```

Example 2.4. Declaring VoiceCall as an Event Type

```
declare VoiceCall
    @role( event )
end
```

@timestamp

A timestamp is automatically assigned to every event. By default, the time is provided by the session clock and assigned to the event at insertion into the working memory. Events can have their own timestamp attribute, which can be included by telling the engine to use the attribute's timestamp instead of the session clock.

To use the attribute's timestamp, use the attribute name as the parameter for the **`@timestamp`** tag.

```
@timestamp( <attributeName> )
```

Example 2.5. Declaring the VoiceCall Timestamp Attribute

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
end
```

@duration

JBoss BRMS Complex Event Processing supports both point-in-time and interval-based events. A point-in-time event is represented as an interval-based event with a duration of zero time units. By default, every event has a duration of zero. To assign a different duration to an event, use the attribute name as the parameter for the **`@duration`** tag.

```
@duration( <attributeName> )
```

Example 2.6. Declaring the VoiceCall Duration Attribute

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
end
```

@expires

Events may be set to expire automatically after a specific duration in the working memory. By default, this happens when the event can no longer match and activate any of the current rules. You can also explicitly define when an event should expire. The @expires tag is only used when the engine is running in *stream* mode.

```
@expires( <timeOffset> )
```

The value of **`timeOffset`** is a temporal interval that sets the relative duration of the event.

```
[#d][#h][#m][#s][#[ms]]
```

All parameters are optional and the # parameter should be replaced by the appropriate value.

To declare that the **`VoiceCall`** facts should expire one hour and thirty-five minutes after insertion into the working memory, use the following:

Example 2.7. Declaring the Expiration Offset for the VoiceCall Events

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
    @expires( 1h35m )
end
```

See Also:

- Section 2.6, “Event Processing Modes”
- Section 2.14.2, “Explicit Expiration”

Report a bug

2.4. SESSION CLOCK

Events have strong temporal constraints making it is necessary to use a reference clock. If a rule needs to determine the average price of a given stock over the last sixty minutes, it is necessary to compare the stock price event's timestamp with the current time. The reference clock provides the current time.

Because the rules engine can simultaneously run an array of different scenarios that require different clocks, multiple clock implementations can be used by the engine.

Scenarios that require different clocks include the following:

- Rules testing: Testing always requires a controlled environment, and when the tests include rules with temporal constraints, it is necessary to control the input rules, facts, and the flow of time.
- Regular execution: A rules engine that reacts to events in real time needs a real-time clock.
- Special environments: Specific environments may have specific time control requirements. For instance, clustered environments may require clock synchronization or JEE environments may require you to use an application server-provided clock.

Report a bug

2.5. AVAILABLE CLOCK IMPLEMENTATIONS

JBoss BRMS Complex Event Processing comes equipped with two clock implementations:

Real-time clock.

This clock is based on the system clock. This is the default.

Psuedo-clock.

This clock is controlled by the application.

Real-Time Clock

The real-time clock is the default. The real-time clock uses the system clock to determine the current time for timestamps.

To explicitly configure the engine to use the real-time clock, set the session configuration parameter to ***realtime***:

```
KnowledgeSessionConfiguration config =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
   config.setOption( ClockTypeOption.get("realtime") );
```

Pseudo-Clock

The pseudo-clock is useful for testing temporal rules since it can be controlled by the application.

To explicitly configure the engine to use the pseudo-clock, set the session configuration parameter to ***pseudo***:

```
KnowledgeSessionConfiguration config =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
   config.setOption( ClockTypeOption.get("pseudo") );
```

This example shows how to control the pseudo-clock:

```
KnowledgeSessionConfiguration conf =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
   conf.setOption( ClockTypeOption.get( "pseudo" ) );
   StatefulKnowledgeSession session = kbase.newStatefulKnowledgeSession(
conf, null );

   SessionPseudoClock clock = session.getSessionClock();

   // then, while inserting facts, advance the clock as necessary:
   FactHandle handle1 = session.insert( tick1 );
   clock.advanceTime( 10, TimeUnit.SECONDS );
   FactHandle handle2 = session.insert( tick2 );
   clock.advanceTime( 30, TimeUnit.SECONDS );
   FactHandle handle3 = session.insert( tick3 );
```

Report a bug

2.6. EVENT PROCESSING MODES

Rules engines process facts and rules to provide applications with results. Regular facts (facts with no temporal constraints) are processed independent of time and in no particular order. JBoss BRMS processes facts of this type in cloud mode. Events (facts which have strong temporal constraints) must be processed in real-time or near real-time. JBoss BRMS processes these events in stream mode. Stream mode deals with synchronization and makes it possible for JBoss BRMS to process events.

Report a bug

2.7. CLOUD MODE

Cloud mode is the default operating mode of JBoss Business Rules Management System.

Running in Cloud mode, the engine applies a many-to-many pattern matching algorithm, which treats the events as an unordered cloud. Events still have timestamps, but there is no way for the rules engine running in Cloud mode to draw relevance from the timestamp, because Cloud mode is unaware of the present time.

This mode uses the rules constraints to find the matching tuples, activate, and fire rules.

Cloud mode does not impose any kind of additional requirements on facts; however, because it has no concept of time, it cannot take advantage of temporal features such as *sliding windows* or *automatic life-cycle management*. In Cloud mode, it is necessary to explicitly retract events when they are no longer needed.

Cloud mode is the default mode. Cloud mode can be specified either by setting a system property, using configuration property files, or via the API.

The API call follows:

```
KnowledgeBaseConfiguration config =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
   config.setOption( EventProcessingOption.CLOUD );
```

The equivalent property follows:

```
drools.eventProcessingMode = cloud
```

Report a bug

2.8. STREAM MODE

Stream mode processes events chronologically as they are inserted into the rules engine. Stream mode uses a session clock that enables the rules engine to process events as they occur in time. The session clock enables processing events as they occur based on the age of the events. Stream mode also synchronizes streams of events (so events in different streams can be processed in chronological order), implements sliding windows of interest, and enables automatic life-cycle management.

The requirements for using stream mode are the following:

- Events in each stream must be ordered chronologically.
- A session clock must be present to synchronize event streams.

NOTE

The application does not need to enforce ordering events between streams, but the use of event streams that have not been synchronized may cause unexpected results.

Stream mode can be enabled by setting a system property, using configuration property files, or via the API.

The API call follows:

```
KnowledgeBaseConfiguration config =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
   config.setOption( EventProcessingOption.STREAM );
```

The equivalent property follows:

```
drools.eventProcessingMode = stream
```

Report a bug

2.9. SUPPORT FOR EVENT STREAMS

Complex event processing use cases deal with streams of events. The streams can be provided to the application via JMS queues, flat text files, database tables, raw sockets, or even web service calls.

Streams share a common set of characteristics:

- Events in the stream are ordered by timestamp. The timestamps may have different semantics for different streams, but they are always ordered internally.
- There is usually a high volume of events in the stream.
- Atomic events contained in the streams are rarely useful by themselves.
- Streams are either homogeneous (they contain a single type of event) or heterogeneous (they contain events of different types).

A stream is also known as an *entry point*.

Facts from one entry point, or stream, may join with facts from any other entry point in addition to facts already in working memory. Facts always remain associated with the entry point through which they entered the engine. Facts of the same type may enter the engine through several entry points, but facts that enter the engine through entry point A will never match a pattern from entry point B.

See Also:

- Section 2.10, “Declaring and Using Entry Points”

Report a bug

2.10. DECLARING AND USING ENTRY POINTS

Entry points are declared implicitly by making direct use of them in rules. Referencing an entry point in a rule will make the engine, at compile time, identify and create the proper internal structures to support that entry point.

For example, a banking application that has transactions fed into the engine via streams could have one stream for all of the transactions executed at ATMs. A rule for this scenario could state, "A withdrawal is only allowed if the account balance is greater than the withdrawal amount the customer has requested."

Example 2.8. Example ATM Rule

```
rule "authorize withdraw"
   when
       WithdrawRequest( $ai : accountId, $am : amount ) from entry-point
"ATM Stream"
       CheckingAccount( accountId == $ai, balance > $am )
   then
       // authorize withdraw
   end
```

When the engine compiles this rule, it will identify that the pattern is tied to the entry point "ATM Stream." The engine will create all the necessary structures for the rule-base to support the "ATM Stream", and this rule will only match **`WithdrawRequest`** events coming from the "ATM Stream."

Note the ATM example rule joins the event (**`WithdrawalRequest`**) from the stream with a fact from the main working memory (**`CheckingAccount`**).

The banking application may have a second rule that states, "A fee of $2 must be applied to a withdraw request made via a branch teller."

Example 2.9. Using Multiple Streams

```
rule "apply fee on withdraws on branches"
   when
       WithdrawRequest( $ai : accountId, processed == true ) from entry-
point "Branch Stream"
       CheckingAccount( accountId == $ai )
   then
       // apply a $2 fee on the account
   end
```

This rule matches events of the same type (**`WithdrawRequest`**) as the example ATM rule but from a different stream. Events inserted into the "ATM Stream" will never match the pattern on the second rule, which is tied to the "Branch Stream;" accordingly, events inserted into the "Branch Stream" will never match the pattern on the example ATM rule, which is tied to the "ATM Stream".

Declaring the stream in a rule states that the rule is only interested in events coming from that stream.

Events can be inserted manually into an entry point instead of directly into the working memory.

Example 2.10. Inserting Facts into an Entry Point

```
// create your rulebase and your session as usual
   StatefulKnowledgeSession session = ...

   // get a reference to the entry point
   WorkingMemoryEntryPoint atmStream =
session.getWorkingMemoryEntryPoint( "ATM Stream" );

   // and start inserting your facts into the entry point
   atmStream.insert( aWithdrawRequest );
```

Applications typically use an adapter to plug a stream entry point, such as a JMS queue, directly into the engine entry point without manually coding the inserts.

Report a bug

2.11. NEGATIVE PATTERN IN STREAM MODE

A *negative pattern* is concerned with conditions that are not met. Negative patterns make reasoning in the absence of events possible. For instance, a safety system could have a rule that states, "If a fire is detected and the sprinkler is *not* activated, sound the alarm."

In Cloud mode, the engine assumes all facts (regular facts and events) are known in advance and evaluates negative patterns immediately.

Example 2.11. A Rule with a Negative Pattern

```
rule "Sound the alarm"
when
```

```
    $f : FireDetected( )
    not( SprinklerActivated( ) )
then
    // sound the alarm
end
```

In stream mode, negative patterns with temporal constraints may force the engine to wait for a set time before activating a rule. A rule may be written for an alarm system that states, "If a fire is detected and the sprinkler is *not* activated after 10 seconds, sound the alarm."

Example 2.12. A Rule with a Negative Pattern with Temporal Constraints

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

Report a bug

2.12. TEMPORAL REASONING

2.12.1. Temporal Reasoning

Complex Event Processing requires the rules engine to engage in temporal reasoning. Events have strong temporal constraints so it is vital the rules engine can determine and interpret an event's temporal attributes, both as they relate to other events and the 'flow of time' as it appears to the rules engine. This makes it possible for rules to take time into account; for instance, a rule could state, "Calculate the average price of a stock over the last 60 minutes."

NOTE

JBoss BRMS Complex Event Processing implements interval-based time events, which have a duration attribute that is used to indicate how long an event is of interest. Point-in-time events are also supported and treated as interval-based events with a duration of 0 (zero).

Report a bug

2.12.2. Temporal Operations

2.12.2.1. Temporal Operations

JBoss BRMS Complex Event Processing implements 13 temporal operators and their logical complements (negation). The 13 temporal operators are the following:

- After
- Before
- Coincides
- During
- Finishes
- Finishes By
- Includes
- Meets
- Met By
- Overlaps
- Overlapped By
- Starts
- Started By

Report a bug

2.12.2.2. After

The **`after`** operator correlates two events and matches when the temporal distance (the time between the two events) from the current event to the event being correlated falls into the distance range declared for the operator.

For example:

```
$eventA : EventA( this after[ 3m30s, 4m ] $eventB )
```

This pattern only matches if the temporal distance between the time when **`$eventB`** finished and the time when **`$eventA`** started is between the lower limit of three minutes and thirty seconds and the upper limit of four minutes.

This can also be represented as follows:

```
  3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

The **`after`** operator accepts one or two optional parameters:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
- If no value is defined, the interval starts at one millisecond and runs indefinitely with no end time.

The **after** operator also accepts negative temporal distances.

For example:

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
```

If the first value is greater than the second value, the engine will automatically reverse them.

The following two patterns are equivalent to each other:

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
   $eventA : EventA( this after[ -2m, -3m30s ] $eventB )
```

Report a bug

2.12.2.3. Before

The **before** operator correlates two events and matches when the temporal distance (time between the two events) from the event being correlated to the current event falls within the distance range declared for the operator.

For example:

```
$eventA : EventA( this before[ 3m30s, 4m ] $eventB )
```

This pattern only matches if the temporal distance between the time when **$eventA** finished and the time when **$eventB** started is between the lower limit of three minutes and thirty seconds and the upper limit of four minutes.

This can also be represented as follows:

```
 3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

The **before** operator accepts one or two optional parameters:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
- If no value is defined, the interval starts at one millisecond and runs indefinitely with no end time.

The **before** operator also accepts negative temporal distances.

For example:

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
```

If the first value is greater than the second value, the engine will automatically reverse them.

The following two patterns are equivalent to each other:

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
   $eventA : EventA( this before[ -2m, -3m30s ] $eventB )
```

Report a bug

2.12.2.4. Coincides

The **`coincides`** operator correlates two events and matches when both events happen at the same time.

For example:

```
$eventA : EventA( this coincides $eventB )
```

This pattern only matches if both the start timestamps of **`$eventA`** and **`$eventB`** are identical and the end timestamps of both **`$eventA`** and **`$eventB`** are also identical.

The **`coincides`** operator accepts optional thresholds for the distance between the events' start times and the events' end times, so the events do not have to start at exactly the same time or end at exactly the same time, but they need to be within the provided thresholds.

The following rules apply when defining thresholds for the **`coincides`** operator:

- If only one parameter is given, it is used to set the threshold for both the start and end times of both events.
- If two parameters are given, the first is used as a threshold for the start time and the second one is used as a threshold for the end time.

For example:

```
$eventA : EventA( this coincides[15s, 10s] $eventB )
```

This pattern will only match if the following conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 15s &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 10s
```

WARNING

The **`coincides`** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance internals.

Report a bug

2.12.2.5. During

The **`during`** operator correlates two events and matches when the current event happens during the event being correlated.

For example:

```
$eventA : EventA( this during $eventB )
```

This pattern only matches if **`$eventA`** starts after **`$eventB`** and ends before **`$eventB`** ends.

This can also be represented as follows:

```
$eventB.startTimestamp < $eventA.startTimestamp <= $eventA.endTimestamp <
$eventB.endTimestamp
```

The **`during`** operator accepts one, two, or four optional parameters:

The following rules apply when providing parameters for the **`during`** operator:

- If one value is defined, this value will represent the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.
- If two values are defined, these values represent a threshold that the current event's start time and end time must occur between in relation to the correlated event's start and end times.

  If the values 5s and 10s are provided, the current event must start between 5 and 10 seconds after the correlated event, and similarly the current event must end between 5 and 10 seconds before the correlated event.
- If four values are defined, the first and second values will be used as the minimum and maximum distances between the starting times of the events, and the third and fourth values will be used as the minimum and maximum distances between the end times of the two events.

Report a bug

2.12.2.6. Finishes

The **`finishes`** operator correlates two events and matches when the current event's start timestamp post-dates the correlated event's start timestamp and both events end simultaneously.

For example:

```
$eventA : EventA( this finishes $eventB )
```

This pattern only matches if **`$eventA`** starts after **`$eventB`** starts and ends at the same time as **`$eventB`** ends.

This can be represented as follows:

```
$eventB.startTimestamp < $eventA.startTimestamp &&
   $eventA.endTimestamp == $eventB.endTimestamp
```

The **`finishes`** operator accepts one optional parameter. If defined, the optional parameter sets the maximum time allowed between the end times of the two events.

For example:

```
$eventA : EventA( this finishes[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
$eventB.startTimestamp < $eventA.startTimestamp &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

WARNING

The **finishes** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

2.12.2.7. Finishes By

The **finishedby** operator correlates two events and matches when the current event's start time predates the correlated event's start time but both events end simultaneously. **finishedby** is the symmetrical opposite of the **finishes** operator.

For example:

```
$eventA : EventA( this finishedby $eventB )
```

This pattern only matches if **$eventA** starts before **$eventB** starts and ends at the same time as **$eventB** ends.

This can be represented as follows:

```
$eventA.startTimestamp < $eventB.startTimestamp &&
   $eventA.endTimestamp == $eventB.endTimestamp
```

The **finishedby** operator accepts one optional parameter. If defined, the optional parameter sets the maximum time allowed between the end times of the two events.

```
$eventA : EventA( this finishedby[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
$eventA.startTimestamp < $eventB.startTimestamp &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

WARNING

The **`finishedby`** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

2.12.2.8. Includes

The **`includes`** operator examines two events and matches when the event being correlated happens during the current event. It is the symmetrical opposite of the **`during`** operator.

For example:

```
$eventA : EventA( this includes $eventB )
```

This pattern only matches if **`$eventB`** starts after **`$eventA`** and ends before **`$eventA`** ends.

This can be represented as follows:

```
$eventA.startTimestamp < $eventB.startTimestamp <= $eventB.endTimestamp <
$eventA.endTimestamp
```

The **`includes`** operator accepts 1, 2 or 4 optional parameters:

- If one value is defined, this value will represent the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.
- If two values are defined, these values represent a threshold that the current event's start time and end time must occur between in relation to the correlated event's start and end times.

  If the values 5s and 10s are provided, the current event must start between 5 and 10 seconds after the correlated event, and similarly the current event must end between 5 and 10 seconds before the correlated event.
- If four values are defined, the first and second values will be used as the minimum and maximum distances between the starting times of the events, and the third and fourth values will be used as the minimum and maximum distances between the end times of the two events.

Report a bug

2.12.2.9. Meets

The **`meets`** operator correlates two events and matches when the current event ends at the same time as the correlated event starts.

For example:

```
$eventA : EventA( this meets $eventB )
```

This pattern matches if **$eventA** ends at the same time as **$eventB** starts.

This can be represented as follows:

```
abs( $eventB.startTimestamp - $eventA.endTimestamp ) == 0
```

The **meets** operator accepts one optional parameter. If defined, it determines the maximum time allowed between the end time of the current event and the start time of the correlated event.

For example:

```
$eventA : EventA( this meets[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventB.startTimestamp - $eventA.endTimestamp) <= 5s
```

WARNING

The **meets** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

2.12.2.10. Met By

The **metby** operator correlates two events and matches when the current event starts at the same time as the correlated event ends.

For example:

```
$eventA : EventA( this metby $eventB )
```

This pattern matches if **$eventA** starts at the same time as **$eventB** ends.

This can be represented as follows:

```
abs( $eventA.startTimestamp - $eventB.endTimestamp ) == 0
```

The **metby** operator accepts one optional parameter. If defined, it sets the maximum distance between the end time of the correlated event and the start time of the current event.

For example:

```
$eventA : EventA( this metby[ 5s ] $eventB )
```

–

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.endTimestamp) <= 5s
```

WARNING

The **`metby`** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

2.12.2.11. Overlaps

The **`overlaps`** operator correlates two events and matches when the current event starts before the correlated event starts and ends after the correlated event starts, but it ends before the correlated event ends.

For example:

```
$eventA : EventA( this overlaps $eventB )
```

This pattern matches if these conditions are met:

```
$eventA.startTimestamp < $eventB.startTimestamp < $eventA.endTimestamp <
$eventB.endTimestamp
```

The **`overlaps`** operator accepts one or two optional parameters:

- If one parameter is defined, it will define the maximum distance between the start time of the correlated event and the end time of the current event.
- If two values are defined, the first value will be the minimum distance, and the second value will be the maximum distance between the start time of the correlated event and the end time of the current event.

Report a bug

2.12.2.12. Overlapped By

The **`overlappedby`** operator correlates two events and matches when the correlated event starts before the current event, and the correlated event ends after the current event starts but before the current event ends.

For example:

```
$eventA : EventA( this overlappedby $eventB )
```

▪

This pattern matches if these conditions are met:

```
$eventB.startTimestamp < $eventA.startTimestamp < $eventB.endTimestamp <
$eventA.endTimestamp
```

The **overlappedby** operator accepts one or two optional parameters:

- If one parameter is defined, it sets the maximum distance between the start time of the correlated event and the end time of the current event.
- If two values are defined, the first value will be the minimum distance, and the second value will be the maximum distance between the start time of the correlated event and the end time of the current event.

Report a bug

2.12.2.13. Starts

The **starts** operator correlates two events and matches when they start at the same time, but the current event ends before the correlated event ends.

For example:

```
$eventA : EventA( this starts $eventB )
```

This pattern matches if **$eventA** and **$eventB** start at the same time, and **$eventA** ends before **$eventB** ends.

This can be represented as follows:

```
$eventA.startTimestamp == $eventB.startTimestamp &&
   $eventA.endTimestamp < $eventB.endTimestamp
```

The **starts** operator accepts one optional parameter. If defined, it determines the maximum distance between the start times of events in order for the operator to still match:

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s &&
   $eventA.endTimestamp < $eventB.endTimestamp
```

WARNING

The **`starts`** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

2.12.2.14. Started By

The **`startedby`** operator correlates two events. It matches when both events start at the same time and the correlating event ends before the current event.

For example:

```
$eventA : EventA( this startedby $eventB )
```

This pattern matches if **`$eventA`** and **`$eventB`** start at the same time, and **`$eventB`** ends before **`$eventA`** ends.

This can be represented as follows:

```
$eventA.startTimestamp == $eventB.startTimestamp &&
   $eventA.endTimestamp > $eventB.endTimestamp
```

The **`startedby`** operator accepts one optional parameter. If defined, it sets the maximum distance between the start time of the two events in order for the operator to still match:

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s &&
   $eventA.endTimestamp > $eventB.endTimestamp
```

WARNING

The **`startsby`** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

2.13. SLIDING TIME WINDOWS

2.13.1. Sliding Time Windows

Stream mode allows events to be matched over a sliding time window. A *sliding window* is a time period that stretches back in time from the present. For instance, a sliding window of two minutes includes any events that have occurred in the past two minutes. As events fall out of the sliding time window (in this case because they occurred more than two minutes ago), they will no longer match against rules using this particular sliding window.

For example:

```
StockTick() over window:time( 2m )
```

JBoss BRMS Complex Event Processing uses the **`over`** keyword to associate windows with patterns.

Sliding time windows can also be used to calculate averages and over time. For instance, a rule could be written that states, "If the average temperature reading for the last ten minutes goes above a certain point, sound the alarm."

Example 2.13. Average Value over Time

```
rule "Sound the alarm in case temperature rises above threshold"
when
    TemperatureThreshold( $max : max )
    Number( doubleValue > $max ) from accumulate(
        SensorReading( $temp : temperature ) over window:time( 10m ),
        average( $temp ) )
then
    // sound the alarm
end
```

The engine will automatically discard any **`SensorReading`** more than ten minutes old and keep re-calculating the average.

Report a bug

2.14. MEMORY MANAGEMENT FOR EVENTS

2.14.1. Memory Management for Events

Automatic memory management for events is available when running the rules engine in Stream mode. Events that no longer match any rule due to their temporal constraints can be safely retracted from the session by the rules engine without any side effects, releasing any resources held by the retracted events.

The rules engine has two ways of determining if an event is still of interest:

Explicitly

Event expiration can be explicitly set with the @expires

Implicitly

The rules engine can analyze the temporal constraints in rules to determine the window of interest for events.

Report a bug

2.14.2. Explicit Expiration

Explicit expiration is set with a **`declare`** statement and the metadata @expires tag.

For example:

Example 2.14. Declaring Explicit Expiration

```
declare StockTick
       @expires( 30m )
   end
```

Declaring expiration against an event-type will, in the above example **`StockTick`** events, remove any StockTick events from the session automatically after the defined expiration time if no rules still need the events.

Report a bug

2.14.3. Inferred Expiration

The rules engine can calculate the expiration offset for a given event implicitly by analyzing the temporal constraints in the rules.

For example:

Example 2.15. A Rule with Temporal Constraints

```
rule "correlate orders"
   when
       $bo : BuyOrder( $id : id )
       $ae : AckOrder( id == $id, this after[0,10s] $bo )
   then
       // do something
   end
```

For the example rule, the rules engine automatically calculates that whenever a **`BuyOrder`** event occurs it needs to store the event for up to ten seconds to wait for the matching **`AckOrder`** event, making the implicit expiration offset for **`BuyOrder`** events ten seconds. An **`AckOrder`** event can only match an existing **`BuyOrder`** event making its implicit expiration offset zero seconds.

The engine analyzes the entire rule-base to find the offset for every event-type. Whenever an implicit expiration clashes with an explicit expiration the engine uses the greater value of the two.

Report a bug

APPENDIX A. REVISION HISTORY

Revision 5.3.1-34.400	**2013-10-31**	**Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 5.3.1-34	**Mon Dec 03 2012**	**L Carlon**

Updated documentation for the JBoss Enterprise BRMS Platform 5.3.1 realease.