Drools Fusion User Guide

Version 6.0.0.CR2

by *The JBoss Drools team* [http://www.jboss.org/drools/team.html]

Chapter 1. Introduction

In the Drools vision of a unified behavioral modelling platform, Drools Fusion is the module responsible for enabling event processing behavior.

1.1. Complex Event Processing

Although several tries were made, there isn't up to date any broadly accepted definition on the term Complex Event Processing. The term Event by itself is frequently overloaded and used to refer to several different things, depending on the context it is used. Defining terms is not the goal of this guide and as so, lets adopt a loose definition that, although not formal, will allow us to proceed with a common understanding.

So, in the scope of this guide:

> **Important**
>
> **Event**, is a record of a significant change of state in the application domain.

For instance, on a Stock Broker application, when a sell operation is executed, it causes a change of state in the domain. This change of state can be observed on several entities in the domain, like the price of the securities that changed to match the value of the operation, the owner of the individual traded assets that change from the seller to the buyer, the balance of the accounts from both seller and buyer that are credited and debited, etc. Depending on how the domain is modelled, this change of state may be represented by a single event, multiple atomic events or even hierarchies of correlated events. In any case, in the context of this guide, Event is the record of the change on a particular data in the domain.

Events are processed by computer systems since they were invented, and throughout the history, systems responsible for that were given different names and different methodologies were employed. It wasn't until the 90's though, that a more focused work started on EDA (Event Driven Architecture) with a more formal definition on the requirements and goals for event processing. Old messaging systems started to change to address such requirements and new systems started to be developed with the single purpose of event processing. Two trends were born under the names of Event Stream Processing and Complex Event Processing.

In the very beginnings, Event Stream Processing was focused on the capabilities of processing streams of events in (near) real time, where the main focus of Complex Event Processing was on the correlation and composition of atomic events into complex (compound) events. An important (maybe the most important) milestone was the publishing of the Dr. David Luckham's book "The Power of Events" in 2002. In the book, Dr Luckham introduces the concept of Complex Event Processing and how it can be used to enhance systems that deal with events. Over the years, both trends converged to a common understanding and today these systems are all referred as CEP systems.

This is a very simplistic explanation to a really complex and fertile field of research, but sets a very high level and common understanding for the concepts this guide will introduce.

The current understanding of what Complex Event Processing is may be briefly described as the following quote from Wikipedia:

> **Important**
>
> "**Complex Event Processing**, or CEP, is primarily an event processing concept that deals with the task of processing multiple events with the goal of identifying the meaningful events within the event cloud. CEP employs techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events such as causality, membership, and timing, and event-driven processes."
>
> —*Wikipedia* [http://en.wikipedia.org/wiki/ Complex_event_processing]

In other words, CEP is about detecting and selecting the interesting events (and only them) from an event cloud, finding their relationships and inferring new data from them and their relationships.

> **Note**
>
> For the remaining of this guide, we will use the terms **Complex Event Processing** and **CEP** as a broad reference for any of the related technologies and techniques, including but not limited to, CEP, Complex Event Processing, ESP, Event Stream Processing and Event Processing in general.

1.2. Drools Fusion

Event Processing use cases, in general, share several requirements and goals with Business Rules use cases. These overlaps happen both on the business side and on the technical side.

On the Business side:

- Business rules are frequently defined based on the occurrence of scenarios triggered by events. Examples could be:

  - On an algorithmic trading application: take an action if the security price increases X% compared to the day opening price, where the price increases are usually denoted by events on a Stock Trade application.

  - On a monitoring application: take an action if the temperature on the server room increases X degrees in Y minutes, where sensor readings are usually denoted by events.

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

From a technical perspective:

- Both require seamless integration with the enterprise infrastructure and applications, specially on autonomous governance, including, but not limited to, lifecycle management, auditing, security, etc.

- Both have functional requirements like pattern matching and non-functional requirements like response time and query/rule explanation.

Even sharing requirements and goals, historically, both fields were born appart and although the industry evolved and one can find good products on the market, they either focus on event processing or on business rules management. That is due not only because of historical reasons but also because, even overlapping in part, use cases do have some different requirements.

> **Important**
>
> Drools was also born as a rules engine several years ago, but following the vision of becoming a single platform for behavioral modelling, it soon realized that it could only achieve this goal by crediting the same importance to the three complementary business modelling techniques:
>
> - Business Rules Management
>
> - Business Processes Management
>
> - Complex Event Processing

In this context, Drools Fusion is the module responsible for adding event processing capabilities into the platform.

Supporting Complex Event Processing, though, is much more than simply understanding what an event is. CEP scenarios share several common and distinguishing characteristics:

- Usually required to process huge volumes of events, but only a small percentage of the events are of real interest.

- Events are usually immutable, since they are a record of state change.

- Usually the rules and queries on events must run in reactive modes, i.e., react to the detection of event patterns.

- Usually there are strong temporal relationships between related events.

- Individual events are usually not important. The system is concerned about patterns of related events and their relationships.

- Usually, the system is required to perform composition and aggregation of events.

Based on this general common characteristics, Drools Fusion defined a set of goals to be achieved in order to support Complex Event Processing appropriately:

- Support Events, with their proper semantics, as first class citizens.

- Allow detection, correlation, aggregation and composition of events.

- Support processing of 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 CEP use cases.

- Support to (re)active rules.

- Support adapters for event input into the engine (pipeline).

The above list of goals are based on the requirements not covered by Drools Expert itself, since in a unified platform, all features of one module are leveraged by the other modules. This way, Drools Fusion is born with enterprise grade features like Pattern Matching, that is paramount to a CEP product, but that is already provided by Drools Expert. In the same way, all features provided by Drools Fusion are leveraged by Drools Flow (and vice-versa) making process management aware of event processing and vice-versa.

For the remaining of this guide, we will go through each of the features Drools Fusion adds to the platform. All these features are available to support different use cases in the CEP world, and the user is free to select and use the ones that will help him model his business use case.

Chapter 2. Drools Fusion Features

2.1. Events

Events, from a Drools perspective are just a special type of fact. In this way, we can say that all events are facts, but not all facts are events. In the next few sections the specific differences that characterize an event are presented.

2.1.1. Event Semantics

An *event* is a fact that present a few distinguishing characteristics:

- **Usually immutables:** since, by the previously discussed definition, events are a record of a state change in the application domain, i.e., a record of something that already happened, and the past can not be "changed", events are immutables. This constraint is an important requirement for the development of several optimizations and for the specification of the event lifecycle. This does not mean that the java object representing the object must be immutable. Quite the contrary, the engine does not enforce immutability of the object model, because one of the most common use cases for rules is event data enrichment.

> **Note**
>
> As a best practice, the application is allowed to populate un-populated event attributes (to enrich the event with inferred data), but already populated attributes should never be changed.

- **Strong temporal constraints:** rules involving events usually require the correlation of multiple events, specially temporal correlations where events are said to happen at some point in time relative to other events.

- **Managed lifecycle:** due to their immutable nature and the temporal constraints, events usually will only match other events and facts during a limited window of time, making it possible for the engine to manage the lifecycle of the events automatically. In other words, one an event is inserted into the working memory, it is possible for the engine to find out when an event can no longer match other facts and automatically retract it, releasing its associated resources.

- **Use of sliding windows:** since all events have timestamps associated to them, it is possible to define and use sliding windows over them, allowing the creation of rules on aggregations of values over a period of time. Example: average of an event value over 60 minutes.

Drools supports the declaration and usage of events with both semantics: **point-in-time** events and **interval-based** events.

> **Note**
>
> A simplistic way to understand the unitification of the semantics is to consider a *point-in-time* event as an *interval-based* event whose *duration is zero.*

2.1.2. Event Declaration

To declare a fact type as an event, all it is required is to assign the @role metadata tag to the fact type. The @role metadata tag accepts two possible values:

- fact : this is the default, declares that the type is to be handled as a regular fact.

- event : declares that the type is to be handled as an event.

For instance, the example below is declaring that the fact type StockTick in a stock broker application shall be handled as an event.

Example 2.1. declaring a fact type as an event

```
import some.package.StockTick


declare StockTick
    @role( event )
end
```

The same applies to facts declared inline. So, if StockTick was a fact type declared in the DRL itself, instead of a previously existing class, the code would be:

Example 2.2. declaring a fact type and assigning it the event role

```
declare StockTick
    @role( event )

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

For more information on type declarations, please check the Rule Language section of the Drools Expert documentation.

2.1.3. Event Metadata

All events have a set of metadata associated to them. Most of the metadata values have defaults that are automatically assigned to each event when they are inserted into the working memory, but it is possible to change the default on an event type basis, using the metadata tags listed below.

For the examples, lets assume the user has the following class in the application domain model:

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
}
```

2.1.3.1. @role

The @role meta data was already discussed in the previous section and is presented here for completeness:

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

It annotates a given fact type as either a regular fact or event. It accepts either "fact" or "event" as a parameter. Default is "fact".

Example 2.4. declaring VoiceCall as an event type

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

2.1.3.2. @timestamp

Every event has an associated timestamp assigned to it. By default, the timestamp for a given event is read from the Session Clock and assigned to the event at the time the event is inserted

into the working memory. Although, sometimes, the event has the timestamp as one of it's own attributes. In this case, the user may tell the engine to use the timestamp from the event's attribute instead of reading it from the Session Clock.

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

To tell the engine what attribute to use as the source of the event's timestamp, just list the attribute name as a parameter to the @timestamp tag.

Example 2.5. declaring the VoiceCall timestamp attribute

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

2.1.3.3. @duration

Drools supports both event semantics: point-in-time events and interval-based events. A point-in-time event is represented as an interval-based event whose duration is zero. By default, all events have duration zero. The user may attribute a different duration for an event by declaring which attribute in the event type contains the duration of the event.

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

So, for our VoiceCall fact type, the declaration would be:

Example 2.6. declaring the VoiceCall duration attribute

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

2.1.3.4. @expires

> **Important**
>
> This tag is only considered when running the engine in STREAM mode. Also, additional discussion on the effects of using this tag is made on the Memory Management section. It is included here for completeness.

Events may be automatically expired after some time in the working memory. Typically this happens when, based on the existing rules in the knowledge base, the event can no longer match and activate any rules. Although, it is possible to explicitly define when an event should expire.

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

The value of *timeOffset* is a temporal interval in the form:

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

Where *[ ]* means an optional parameter and *#* means a numeric value.

So, to declare that the VoiceCall facts should be expired after 1 hour and 35 minutes after they are inserted into the working memory, the user would write:

Example 2.7. declaring the expiration offset for the VoiceCall events

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

The @expires policy will take precedence and override the implicit expiration offset calculated from temporal constraints and sliding windows in the knowledge base.

2.2. Session Clock

Reasoning over time requires a reference clock. Just to mention one example, if a rule reasons over the average price of a given stock over the last 60 minutes, how the engine knows what stock price changes happened over the last 60 minutes in order to calculate the average? The obvious

response is: by comparing the timestamp of the events with the "current time". How the engine knows what **time is now**? Again, obviously, by querying the Session Clock.

The session clock implements a strategy pattern, allowing different types of clocks to be plugged and used by the engine. This is very important because the engine may be running in an elements of different scenarios that may require different clock implementations. Just to mention a few:

- **Rules testing:** testing always requires a controlled environment, and when the tests include rules with temporal constraints, it is necessary to not only control the input rules and facts, but also the flow of time.

- **Regular execution:** usually, when running rules in production, the application will require a real time clock that allows the rules engine to react immediately to the time progression.

- **Special environments:** specific environments may have specific requirements on time control. Cluster environments may require clock synchronization through heart beats, or JEE environments may require the use of an AppServer provided clock, etc.

- **Rules replay or simulation:** to replay scenarios or simulate scenarios it is necessary that the application also controls the flow of time.

2.2.1. Available Clock Implementations

Drools 5 provides 2 clock implementations out of the box. The default real time clock, based on the system clock, and an optional pseudo clock, controlled by the application.

2.2.1.1. Real Time Clock

By default, Drools uses a real time clock implementation that internally uses the system clock to determine the current timestamp.

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

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

2.2.1.2. Pseudo Clock

Drools also offers out of the box an implementation of a clock that is controlled by the application that is called Pseudo Clock. This clock is specially useful for unit testing temporal rules since it can be controlled by the application and so the results become deterministic.

To configure the pseudo session clock, do:

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

As an example of how to control the pseudo session 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 );
```

2.3. Streams Support

Most CEP use cases have to deal with streams of events. The streams can be provided to the application in various forms, from JMS queues to flat text files, from database tables to raw sockets or even through web service calls. In any case, the streams share a common set of characteristics:

- events in the stream are ordered by a timestamp. The timestamp may have different semantics for different streams but they are always ordered internally.

- volumes of events are usually high.

- atomic events are rarely useful by themselves. Usually meaning is extracted from the correlation between multiple events from the stream and also from other sources.

- streams may be homogeneous, i.e. contain a single type of events, or heterogeneous, i.e. contain multiple types of events.

Drools generalized the concept of a stream as an "entry point" into the engine. An entry point is for drools a gate from which facts come. The facts may be regular facts or special facts like events.

In Drools, facts from one entry point (stream) may join with facts from any other entry point or event with facts from the working memory. Although, they never mix, i.e., they never lose the reference to the entry point through which they entered the engine. This is important because one may have the same type of facts coming into the engine through several entry points, but one fact that is inserted into the engine through entry point A will never match a pattern from a entry point B, for example.

2.3.1. Declaring and Using Entry Points

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

So, for instance, lets imagine a banking application, where transactions are fed into the system coming from streams. One of the streams contains all the transactions executed in ATM machines. So, if one of the rules says: a withdraw is authorized if and only if the account balance is over the requested withdraw amount, the rule would look like:

Example 2.8. Example of Stream Usage

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

In the previous example, the engine compiler will identify that the pattern is tied to the entry point "ATM Stream" and will both create all the necessary structures for the rulebase to support the "ATM Stream" and will only match WithdrawRequests coming from the "ATM Stream". In the previous example, the rule is also joining the event from the stream with a fact from the main working memory (CheckingAccount).

Now, lets imagine a second rule that states that a fee of $2 must be applied to any account for which a withdraw request is placed at a bank branch:

Example 2.9. Using a different Stream

```
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
```

The previous rule will match events of the exact same type as the first rule (WithdrawRequest), but from two different streams, so an event inserted into "ATM Stream" will never be evaluated

against the pattern on the second rule, because the rule states that it is only interested in patterns coming from the "Branch Stream".

So, entry points, besides being a proper abstraction for streams, are also a way to scope facts in the working memory, and a valuable tool for reducing cross products explosions. But that is a subject for another time.

Inserting events into an entry point is equally simple. Instead of inserting events directly into the working memory, insert them into the entry point as shown in the example below:

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 );
```

The previous example shows how to manually insert facts into a given entry point. Although, usually, the application will use one of the many adapters to plug a stream end point, like a JMS queue, directly into the engine entry point, without coding the inserts manually. The Drools pipeline API has several adapters and helpers to do that as well as examples on how to do it.

2.4. Temporal Reasoning

Temporal reasoning is another requirement of any CEP system. As discussed previously, one of the distinguishing characteristics of events is their strong temporal relationships.

Temporal reasoning is an extensive field of research, from its roots on Temporal Modal Logic to its more practical applications in business systems. There are hundreds of papers and thesis written and approaches are described for several applications. Drools once more takes a pragmatic and simple approach based on several sources, but specially worth noting the following papers:

[ALLEN81] Allen, J.F.. *An Interval-based Representation of Temporal Knowledge*. 1981.

[ALLEN83] Allen, J.F.. *Maintaining knowledge about temporal intervals*. 1983.

[BENNE00] by Bennet, Brandon and Galton, Antony P.. *A Unifying Semantics for Time and Events*. 2005.

[YONEK05] by Yoneki, Eiko and Bacon, Jean. *Unified Semantics for Event Correlation Over Time and Space in Hybrid Network Environments*. 2005.

Drools implements the Interval-based Time Event Semantics described by Allen, and represents Point-in-Time Events as Interval-based evens with duration 0 (zero).

> **Note**
>
> For all temporal operator intervals, the "*" (star) symbol is used to indicate *positive infinity* and the "-*" (minus star) is used to indicate *negative infinity*.

2.4.1. Temporal Operators

Drools implements all 13 operators defined by Allen and also their logical complement (negation). This section details each of the operators and their parameters.

2.4.1.1. After

The after evaluator correlates two events and matches when the temporal distance from the current event to the event being correlated belongs to the distance range declared for the operator.

Lets look at an example:

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

The previous pattern will match if and only if the temporal distance between the time when $eventB finished and the time when $eventA started is between (3 minutes and 30 seconds) and (4 minutes). In other words:

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

The temporal distance interval for the after operator is optional:

- If two values are defined (like in the example below), the interval starts on the first value and finishes on the second.

- If only one value is defined, the interval starts on the value and finishes on the positive infinity.

- If no value is defined, it is assumed that the initial value is 1ms and the final value is the positive infinity.

> **Note**
>
> It is possible to define negative distances for this operator. Example:

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

> **Note**
>
> If the first value is greater than the second value, the engine automatically reverses them, as there is no reason to have the first value greater than the second value. Example: the following two patterns are considered to have the same semantics:
>
> ```
> $eventA : EventA( this after[ -3m30s, -2m ] $eventB )
> $eventA : EventA( this after[ -2m, -3m30s ] $eventB )
> ```

> **Note**
>
> The *after*, *before* and *coincides* operators can be used to define constraints between events, java.util.Date attributes, and long attributes (interpreted as timestamps since epoch) in any combination. Example:
>
> ```
> EventA( this after $someDate )
> ```

2.4.1.2. Before

The before evaluator correlates two events and matches when the temporal distance from the event being correlated to the current correlated belongs to the distance range declared for the operator.

Lets look at an example:

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

The previous pattern will match if and only if the temporal distance between the time when $eventA finished and the time when $eventB started is between (3 minutes and 30 seconds) and (4 minutes). In other words:

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

The temporal distance interval for the before operator is optional:

- If two values are defined (like in the example below), the interval starts on the first value and finishes on the second.

- If only one value is defined, then the interval starts on the value and finishes on the positive infinity.

- If no value is defined, it is assumed that the initial value is 1ms and the final value is the positive infinity.

> **i** **Note**
>
> It is possible to define negative distances for this operator. Example:
>
> ```
> $eventA : EventA( this before[ -3m30s, -2m ] $eventB )
> ```

> **i** **Note**
>
> If the first value is greater than the second value, the engine automatically reverses them, as there is no reason to have the first value greater than the second value. Example: the following two patterns are considered to have the same semantics:
>
> ```
> $eventA : EventA( this before[ -3m30s, -2m ] $eventB )
> $eventA : EventA( this before[ -2m, -3m30s ] $eventB )
> ```

> **i** **Note**
>
> The *after*, *before* and *coincides* operators can be used to define constraints between events, java.util.Date attributes, and long attributes (interpreted as timestamps since epoch) in any combination. Example:
>
> ```
> EventA( this after $someDate )
> ```

2.4.1.3. Coincides

The coincides evaluator correlates two events and matches when both happen at the same time. Optionally, the evaluator accept thresholds for the distance between events' start and finish timestamps.

Lets look at an example:

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

The previous pattern will match if and only if the start timestamps of both $eventA and $eventB are the same AND the end timestamp of both $eventA and $eventB also are the same.

Optionally, this operator accepts one or two parameters. These parameters are the thresholds for the distance between matching timestamps.

- If only one parameter is given, it is used for both start and end timestamps.

- If two parameters are given, then the first is used as a threshold for the start timestamp and the second one is used as a threshold for the end timestamp.

In other words:

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

Above pattern will match if and only if:

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

> ⚠️ **Warning**
>
> It makes no sense to use negative interval values for the parameters and the engine will raise an error if that happens.

> ℹ️ **Note**
>
> The *after*, *before* and *coincides* operators can be used to define constraints between events, java.util.Date attributes, and long attributes (interpreted as timestamps since epoch) in any combination. Example:
>
> ```
> EventA( this after $someDate )
> ```

2.4.1.4. During

The during evaluator correlates two events and matches when the current event happens during the occurrence of the event being correlated.

Lets look at an example:

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

The previous pattern will match if and only if the $eventA starts after $eventB starts and finishes before $eventB finishes.

In other words:

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

The during operator accepts 1, 2 or 4 optional parameters as follow:

- If one value is defined, this will be the maximum distance between the start timestamp of both event and the maximum distance between the end timestamp of both events in order to operator match. Example:

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

Will match if and only if:

```
0 < $eventA.startTimestamp - $eventB.startTimestamp <= 5s &&
0 < $eventB.endTimestamp - $eventA.endTimestamp <= 5s
```

- If two values are defined, the first value will be the minimum distance between the timestamps of both events, while the second value will be the maximum distance between the timestamps of both events. Example:

```
$eventA : EventA( this during[ 5s, 10s ] $eventB )
```

Will match if and only if:

```
5s <= $eventA.startTimestamp - $eventB.startTimestamp <= 10s &&
```

```
5s <= $eventB.endTimestamp - $eventA.endTimestamp <= 10s
```

- If four values are defined, the first two values will be the minimum and maximum distances between the start timestamp of both events, while the last two values will be the minimum and maximum distances between the end timestamp of both events. Example:

```
$eventA : EventA( this during[ 2s, 6s, 4s, 10s ] $eventB )
```

Will match if and only if:

```
2s <= $eventA.startTimestamp - $eventB.startTimestamp <= 6s &&
4s <= $eventB.endTimestamp - $eventA.endTimestamp <= 10s
```

2.4.1.5. Finishes

The finishes evaluator correlates two events and matches when the current event's start timestamp happens after the correlated event's start timestamp, but both end timestamps occur at the same time.

Lets look at an example:

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

The previous pattern will match if and only if the $eventA starts after $eventB starts and finishes at the same time $eventB finishes.

In other words:

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

The finishes evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the end timestamp of both events in order for the operator to match. Example:

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

Will match if and only if:

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

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

> ⚠️ **Warning**
>
> It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

2.4.1.6. Finished By

The finishedby evaluator correlates two events and matches when the current event start timestamp happens before the correlated event start timestamp, but both end timestamps occur at the same time. This is the symmetrical opposite of finishes evaluator.

Lets look at an example:

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

The previous pattern will match if and only if the $eventA starts before $eventB starts and finishes at the same time $eventB finishes.

In other words:

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

The finishedby evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the end timestamp of both events in order for the operator to match. Example:

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

Will match if and only if:

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

> ⚠️ **Warning**
>
> It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

2.4.1.7. Includes

The includes evaluator correlates two events and matches when the event being correlated happens during the current event. It is the symmetrical opposite of during evaluator.

Lets look at an example:

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

The previous pattern will match if and only if the $eventB starts after $eventA starts and finishes before $eventA finishes.

In other words:

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

The includes operator accepts 1, 2 or 4 optional parameters as follow:

- If one value is defined, this will be the maximum distance between the start timestamp of both event and the maximum distance between the end timestamp of both events in order to operator match. Example:

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

Will match if and only if:

```
0 < $eventB.startTimestamp - $eventA.startTimestamp <= 5s &&
0 < $eventA.endTimestamp - $eventB.endTimestamp <= 5s
```

- If two values are defined, the first value will be the minimum distance between the timestamps of both events, while the second value will be the maximum distance between the timestamps of both events. Example:

```
$eventA : EventA( this includes[ 5s, 10s ] $eventB )
```

Will match if and only if:

```
5s <= $eventB.startTimestamp - $eventA.startTimestamp <= 10s &&
5s <= $eventA.endTimestamp - $eventB.endTimestamp <= 10s
```

- If four values are defined, the first two values will be the minimum and maximum distances between the start timestamp of both events, while the last two values will be the minimum and maximum distances between the end timestamp of both events. Example:

```
$eventA : EventA( this includes[ 2s, 6s, 4s, 10s ] $eventB )
```

Will match if and only if:

```
2s <= $eventB.startTimestamp - $eventA.startTimestamp <= 6s &&
4s <= $eventA.endTimestamp - $eventB.endTimestamp <= 10s
```

2.4.1.8. Meets

The meets evaluator correlates two events and matches when the current event's end timestamp happens at the same time as the correlated event's start timestamp.

Lets look at an example:

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

The previous pattern will match if and only if the $eventA finishes at the same time $eventB starts.

In other words:

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

The meets evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the end timestamp of current event and the start timestamp of the correlated event in order for the operator to match. Example:

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

Will match if and only if:

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

> ⚠️ **Warning**
>
> It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

2.4.1.9. Met By

The metby evaluator correlates two events and matches when the current event's start timestamp happens at the same time as the correlated event's end timestamp.

Lets look at an example:

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

The previous pattern will match if and only if the $eventA starts at the same time $eventB finishes.

In other words:

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

The metby evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the end timestamp of the correlated event and the start timestamp of the current event in order for the operator to match. Example:

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

Will match if and only if:

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

> **⚠ Warning**
>
> It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

2.4.1.10. Overlaps

The overlaps evaluator correlates two events and matches when the current event starts before the correlated event starts and finishes after the correlated event starts, but before the correlated event finishes. In other words, both events have an overlapping period.

Lets look at an example:

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

The previous pattern will match if and only if:

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

The overlaps operator accepts 1 or 2 optional parameters as follow:

- If one parameter is defined, this will be the maximum distance between the start timestamp of the correlated event and the end timestamp of the current event. Example:

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

  Will match if and only if:

  ```
  $eventA.startTimestamp < $eventB.startTimestamp < $eventA.endTimestamp <
   $eventB.endTimestamp &&
  0 <= $eventA.endTimestamp - $eventB.startTimestamp <= 5s
  ```

- 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 timestamp of the correlated event and the end timestamp of the current event. Example:

  ```
  $eventA : EventA( this overlaps[ 5s, 10s ] $eventB )
  ```

Will match if and only if:

```
$eventA.startTimestamp  <  $eventB.startTimestamp  <  $eventA.endTimestamp  <
 $eventB.endTimestamp &&
5s <= $eventA.endTimestamp - $eventB.startTimestamp <= 10s
```

2.4.1.11. Overlapped By

The overlappedby evaluator correlates two events and matches when the correlated event starts before the current event starts and finishes after the current event starts, but before the current event finishes. In other words, both events have an overlapping period.

Lets look at an example:

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

The previous pattern will match if and only if:

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

The overlappedby operator accepts 1 or 2 optional parameters as follow:

- If one parameter is defined, this will be the maximum distance between the start timestamp of the current event and the end timestamp of the correlated event. Example:

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

Will match if and only if:

```
$eventB.startTimestamp  <  $eventA.startTimestamp  <  $eventB.endTimestamp  <
 $eventA.endTimestamp &&
0 <= $eventB.endTimestamp - $eventA.startTimestamp <= 5s
```

- 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 timestamp of the current event and the end timestamp of the correlated event. Example:

```
$eventA : EventA( this overlappedby[ 5s, 10s ] $eventB )
```

Will match if and only if:

```
$eventB.startTimestamp  <  $eventA.startTimestamp  <  $eventB.endTimestamp  <
 $eventA.endTimestamp &&
5s <= $eventB.endTimestamp - $eventA.startTimestamp <= 10s
```

2.4.1.12. Starts

The starts evaluator correlates two events and matches when the current event's end timestamp happens before the correlated event's end timestamp, but both start timestamps occur at the same time.

Lets look at an example:

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

The previous pattern will match if and only if the $eventA finishes before $eventB finishes and starts at the same time $eventB starts.

In other words:

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

The starts evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the start timestamp of both events in order for the operator to match. Example:

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

Will match if and only if:

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

> **⚠ Warning**
>
> It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

2.4.1.13. Started By

The startedby evaluator correlates two events and matches when the correlating event's end timestamp happens before the current event's end timestamp, but both start timestamps occur at the same time. Lets look at an example:

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

The previous pattern will match if and only if the $eventB finishes before $eventA finishes and starts at the same time $eventB starts.

In other words:

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

The startedby evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the start timestamp of both events in order for the operator to match. Example:

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

Will match if and only if:

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

> **⚠ Warning**
>
> It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

2.5. Event Processing Modes

Rules engines in general have a well known way of processing data and rules and provide the application with the results. Also, there is not many requirements on how facts should be presented to the rules engine, specially because in general, the processing itself is time independent. That is a good assumption for most scenarios, but not for all of them. When the requirements include the processing of real time or near real time events, time becomes and important variable of the reasoning process.

The following sections will explain the impact of time on rules reasoning and the two modes provided by Drools for the reasoning process.

2.5.1. Cloud Mode

The CLOUD processing mode is the default processing mode. Users of rules engine are familiar with this mode because it behaves in exactly the same way as any pure forward chaining rules engine, including previous versions of Drools.

When running in CLOUD mode, the engine sees all facts in the working memory, does not matter if they are regular facts or events, as a whole. There is no notion of flow of time, although events have a timestamp as usual. In other words, although the engine knows that a given event was created, for instance, on January 1st 2009, at 09:35:40.767, it is not possible for the engine to determine how "old" the event is, because there is no concept of "now".

In this mode, the engine will apply its usual many-to-many pattern matching algorithm, using the rules constraints to find the matching tuples, activate and fire rules as usual.

This mode does not impose any kind of additional requirements on facts. So for instance:

- There is no notion of time. No requirements clock synchronization.

- There is no requirement on event ordering. The engine looks at the events as an unordered cloud against which the engine tries to match rules.

On the other hand, since there is no requirements, some benefits are not available either. For instance, in CLOUD mode, it is not possible to use sliding windows, because sliding windows are based on the concept of "now" and there is no concept of "now" in CLOUD mode.

Since there is no ordering requirement on events, it is not possible for the engine to determine when events can no longer match and as so, there is no automatic life-cycle management for events. I.e., the application must explicitly retract events when they are no longer necessary, in the same way the application does with regular facts.

Cloud mode is the default execution mode for Drools, but in any case, as any other configuration in Drools, it is possible to change this behavior either by setting a system property, using configuration property files or using the API. The corresponding property is:

```
KnowledgeBaseConfiguration config = KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
```

```
config.setOption( EventProcessingOption.CLOUD );
```

The equivalent property is:

```
drools.eventProcessingMode = cloud
```

2.5.2. Stream Mode

The STREAM processing mode is the mode of choice when the application needs to process streams of events. It adds a few common requirements to the regular processing, but enables a whole lot of features that make stream event processing a lot simpler.

The main requirements to use STREAM mode are:

- Events in each stream must be time-ordered. I.e., inside a given stream, events that happened first must be inserted first into the engine.

- The engine will force synchronization between streams through the use of the session clock, so, although the application does not need to enforce time ordering between streams, the use of non-time-synchronized streams may result in some unexpected results.

Given that the above requirements are met, the application may enable the STREAM mode using the following API:

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

Or, the equivalent property:

```
drools.eventProcessingMode = stream
```

When using the STREAM, the engine knows the concept of flow of time and the concept of "now", i.e., the engine understands how old events are based on the current timestamp read from the Session Clock. This characteristic allows the engine to provide the following additional features to the application:

- Sliding Window support

- Automatic Event Lifecycle Management

- Automatic Rule Delaying when using Negative Patterns

All these features are explained in the following sections.

2.5.2.1. Role of Session Clock in Stream mode

When running the engine in CLOUD mode, the session clock is used only to time stamp the arriving events that don't have a previously defined timestamp attribute. Although, in STREAM mode, the Session Clock assumes an even more important role.

In STREAM mode, the session clock is responsible for keeping the current timestamp, and based on it, the engine does all the temporal calculations on event's aging, synchronizes streams from multiple sources, schedules future tasks and so on.

Check the documentation on the Session Clock section to know how to configure and use different session clock implementations.

2.5.2.2. Negative Patterns in Stream Mode

Negative patterns behave different in STREAM mode when compared to CLOUD mode. In CLOUD mode, the engine assumes that all facts and events are known in advance (there is no concept of flow of time) and so, negative patterns are evaluated immediately.

When running in STREAM mode, negative patterns with temporal constraints may require the engine to wait for a time period before activating a rule. The time period is automatically calculated by the engine in a way that the user does not need to use any tricks to achieve the desired result.

For instance:

Example 2.11. a rule that activates immediately upon matching

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( ) )
then
    // sound the alarm
end
```

The above rule has no temporal constraints that would require delaying the rule, and so, the rule activates immediately. The following rule on the other hand, must wait for 10 seconds before activating, since it may take up to 10 seconds for the sprinklers to activate:

Example 2.12. a rule that automatically delays activation due to temporal constraints

```
rule "Sound the alarm"
when
    $f : FireDetected( )
```

```
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

This behaviour allows the engine to keep consistency when dealing with negative patterns and temporal constraints at the same time. The above would be the same as writing the rule as below, but does not burden the user to calculate and explicitly write the appropriate duration parameter:

Example 2.13. same rule with explicit duration parameter

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

The following rule expects every 10 seconds at least one "Heartbeat" event, if not the rule fires. The special case in this rule is that we use the same type of the object in the first pattern and in the negative pattern. The negative pattern has the temporal constraint to wait between 0 to 10 seconds before firing and it excludes the Heartbeat bound to $h. Excluding the bound Heartbeat is important since the temporal constraint [0s, ...] does not exclude by itself the bound event $h from being matched again, thus preventing the rule to fire.

Example 2.14. excluding bound events in negative patterns

```
rule "Sound the alarm"
when
    $h: Heartbeat( ) from entry-point "MonitoringStream"
     not( Heartbeat( this != $h, this after[0s,10s] $h ) from entry-point
 "MonitoringStream" )
then
    // Sound the alarm
end
```

2.6. Sliding Windows

Sliding Windows are a way to scope the events of interest by defining a window that is constantly moving. The two most common types of sliding window implementations are time based windows and length based windows.

The next sections will detail each of them.

> **Important**
>
> Sliding Windows are only available when running the engine in STREAM mode. Check the Event Processing Mode section for details on how the STREAM mode works.

> **Important**
>
> Sliding windows start to match immediately and defining a sliding window does not imply that the rule has to wait for the sliding window to be "full" in order to match. For instance, a rule that calculates the average of an event property on a window:length(10) will start calculating the average immediately, and it will start at 0 (zero) for no-events, and will update the average as events arrive one by one.

2.6.1. Sliding Time Windows

Sliding Time Windows allow the user to write rules that will only match events occurring in the last X time units.

For instance, if the user wants to consider only the Stock Ticks that happened in the last 2 minutes, the pattern would look like this:

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

Drools uses the "over" keyword to associate windows to patterns.

On a more elaborate example, if the user wants to sound an alarm in case the average temperature over the last 10 minutes read from a sensor is above the threshold value, the rule would look like:

Example 2.15. aggregating values over time windows

```
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 disregard any SensorReading older than 10 minutes and keep the calculated average consistent.

> **Important**
>
> Please note that time based windows are considered when calculating the interval an event remains in the working memory before being expired, but an event falling off a sliding window does not mean by itself that the event will be discarded from the working memory, as there might be other rules that depend on that event. The engine will discard events only when no other rules depend on that event and the expiration policy for that event type is fulfilled.

2.6.2. Sliding Length Windows

Sliding Length Windows work the same way as Time Windows, but consider events based on order of their insertion into the session instead of flow of time.

For instance, if the user wants to consider only the last 10 RHT Stock Ticks, independent of how old they are, the pattern would look like this:

```
StockTick( company == "RHT" ) over window:length( 10 )
```

As you can see, the pattern is similar to the one presented in the previous section, but instead of using window:time to define the sliding window, it uses window:length.

Using a similar example to the one in the previous section, if the user wants to sound an alarm in case the average temperature over the last 100 readings from a sensor is above the threshold value, the rule would look like:

Example 2.16. aggregating values over length windows

```
rule "Sound the alarm in case temperature rises above threshold"
when
    TemperatureThreshold( $max : max )
    Number( doubleValue > $max ) from accumulate(
        SensorReading( $temp : temperature ) over window:length( 100 ),
        average( $temp ) )
then
    // sound the alarm
end
```

The engine will keep only consider the last 100 readings to calculate the average temperature.

Important

Please note that falling off a length based window is not criteria for event expiration in the session. The engine disregards events that fall off a window when calculating that window, but does not remove the event from the session based on that condition alone as there might be other rules that depend on that event.

Important

Please note that length based windows do not define temporal constraints for event expiration from the session, and the engine will not consider them. If events have no other rules defining temporal constraints and no explicit expiration policy, the engine will keep them in the session indefinitely.

2.7. KnowledgeBase Partitioning

Warning

THIS FEATURE IS NO LONGER SUPPORTED. It may be reintroduced in the future, but trying to use it in the current version will raise an exception.

The classic Rete algorithm is usually executed using a single thread. Although, as confirmed in several opportunities by Dr. Forgy, the algorithm itself is parallelizable. Drools implementation of the ReteOO algorithm supports coarse grained parallelization through rulebase partitioning.

When this option is enabled, the rulebase will be partitioned in several independent partitions and a pool of worker threads will be used to propagate facts through the partitions. The implementation guarantees that at most one worker thread will be executing tasks for a given partition, but multiple partitions may be "active" at a single point in time.

Everything should be transparent to the user, except that all working memory actions (insert/retract/modify) are executed asynchronously.

Important

This feature enables parallel LHS evaluation, but does not change the behavior of rule firing. I.e., rules will continue to fire sequentially, according to the conflict resolution strategy.

2.7.1. When partitioning is useful

Knowledge base partitioning is a very powerful feature for specific scenarios, but it is not a general case solution. To understand if this feature would be useful for a given scenario, the user may follow the checklist below:

1. Does your hardware contains multiple processors?

2. Does your knowledge session process a high volume of facts?

3. Are the LHS of your rules expensive to evaluate? (ex: use expensive "from" expressions)

4. Does your knowledge base contains hundreds or more rules?

If the answer to all the questions above is "yes", then this feature will probably increase the overall performance of your rulebase evaluation.

2.7.2. How to configure partitioning

To enable knowledge base partitioning, set the following option:

Example 2.17. Enabling multithread evaluation (partitioning)

```
KnowledgeBaseConfiguration config = KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
config.setOption( MultithreadEvaluationOption.YES );
```

The equivalent property is:

```
drools.multithreadEvaluation = <true|false>
```

The default value for this option is "false" (disabled).

2.7.3. Multithreading management

Drools offers a simple configuration option for users to control the size of the worker thread's pool.

To define the maximum size for the thread pool, the user may use the following configuration option:

Example 2.18. setting the maximum number of threads for rule evaluation to 5

```
KnowledgeBaseConfiguration config = KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
config.setOption( MaxThreadsOption.get(5) );
```

The equivalent property is:

```
drools.maxThreads = <-1|1..n>
```

The default value for this configuration is 3 and a negative number means the engine will try to spawn as many threads as there are partitions in the rulebase.

> **Warning**
>
> It is usually dangerous to set this option with a negative number. Always set it with a sensible positive number of threads.

2.8. Memory Management for Events

> **Important**
>
> The automatic memory management for events is only performed when running the engine in STREAM mode. Check the Event Processing Mode section for details on how the STREAM mode works.

One of the benefits of running the engine in STREAM mode is that the engine can detect when an event can no longer match any rule due to its temporal constraints. When that happens, the engine can safely retract the event from the session without side effects and release any resources used by that event.

There are basically 2 ways for the engine to calculate the matching window for a given event:

- explicitly, using the expiration policy

- implicitly, analyzing the temporal constraints on events

2.8.1. Explicit expiration offset

The first way of allowing the engine to calculate the window of interest for a given event type is by explicitly setting it. To do that, just use the declare statement and define an expiration for the fact type:

Example 2.19. explicitly defining an expiration offset of 30 minutes for StockTick events

```
declare StockTick
```

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

The above example declares an expiration offset of 30 minutes for StockTick events. After that time, assuming no rule still needs the event, the engine will expire and remove the event from the session automatically.

2.8.2. Inferred expiration offset

Another way for the engine to calculate the expiration offset for a given event is implicitly, by analyzing the temporal constraints in the rules. For instance, given the following rule:

Example 2.20. example rule with temporal constraints

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

Analyzing the above rule, the engine automatically calculates that whenever a BuyOrderEvent matches, it needs to store it for up to 10 seconds to wait for matching AckEvent's. So, the implicit expiration offset for BuyOrderEvent will be 10 seconds. AckEvent, on the other hand, can only match existing BuyOrderEvent's, and so its expiration offset will be zero seconds.

The engine will make this analysis for the whole rulebase and find the offset for every event type. Whenever an implicit expiration offset clashes with the explicit expiration offset, then engine will use the greater of the two.

Chapter 3. References