Drools Fusion User Guide

Drools Fusion

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

Before we examine Drools Fusion, we need to define some terms. The terms "Event" and "Complex Event Processing" have multiple, overloaded, and sometimes confusing definitions. The goal of this guide is not to attempt to unify all the competing definitions for these terms, but before we start describing Complex Event Processing, we need to define, in informal terms, just what we mean when the guide refers to an "Event."

In the scope of this guide:

Important

Event, is a record of a significant change of state in the application domain. In this context, "domain" refers to the set of data, conditions, and the environment in which the application is running.

For example, in a stock brokerage application, when a sell operation is executed, the operation causes a change of state in the domain. This change of state can be observed on several entities in the domain, such as 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 modeled, this change of state may be represented by a single event, multiple independent (or "atomic") events or even hierarchies of correlated events. In this guide, the term "event" refers to the record of the change on a particular piece or pieces of data within the domain.

Events have been processed by computer systems since they were first invented. Throughout computer and software the history, systems responsible for event processing have referred to event processing with different names and and have used different methodologies. It wasn't until the 1990's however, that more focused work started on **Event Driven Architecture** (EDA) with a more formal definition on the requirements and goals for event processing. Older messaging systems started to change to address these requirements and new systems started to be developed with the single purpose of event processing. Two approaches for event processing developed: **Event Stream Processing** (ESP) and **Complex Event Processing** (CEP).

In the begining, Event Stream Processing was focused on the capabilities of processing streams of events in (near) real time, while, in contrast, the main focus of Complex Event Processing was on the correlation and composition of atomic events into complex (compound) events. An important milestone in the development of event processing was the publication of Dr. David Luckham's book The Power of Events in 2002. In this book, Dr Luckham introduced the concept of "Complex

Event Processing" and how it could be used to enhance systems that deal with events. Over the years, both approaches have converged into a common understanding and today systems implementing either approach are all referred as "CEP systems."

A brief definition of Complex Event Processing is:

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 these events) 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 overlapping characteristics 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. Some examples are:
    - In an algorithmic trading application: perform an action if the security price increases X% compared to the day opening price, where the price increases are usually denoted by events from a stock trade application.

  - In a monitoring application: perform an action if the temperature on a 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 responses for the business to adapt itself to new market conditions, new regulations and new enterprise policies, or a system to addapt to changes in its operational environment..

From a technical perspective:

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

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

Even though they share requirements and goals, historically, Event Processing and Business Rules were separate disciplines. Implementations of each tend to focus on on one discipline or the other.

Important

JBoss Drools was created as a rules engine several years ago. However, to become a single platform for behavioral modelling, it assigns the same importance to these three complementary business modelling techniques:

- Business Rules Management (BRM)

- Business Processes Management (BPM)

- Complex Event Processing (CEP)

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

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

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

- There are usually strong temporal relationships between related events.
- Individual events are usually not important. The system is concerned about patterns of related events and their relationships.
- The system is usually required to perform composition and aggregation of events.

Based on these common characteristics, Drools Fusion has defined a set of requirements to be fulfilled in order to support Complex Event Processing appropriately:

- Support Events, with their propper 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 (re)active rules.
- Support adapters for event input into the engine (pipeline).

In the Drools unified platform, all features of one module are leveraged by the other modules. This means that while the above of requirements that are not fulfilled by Drools Expert, Drools Fusion is able to take advantage of Drools Expert's features such as like Pattern Matching. 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.

In remainder of this guide, we will describe each of the features Drools Fusion adds to the Drools unified platform. All these features are available to support different use cases in the CEP world, and you are free to select and use the ones that will help you model your business use cases.

Chapter 2. Drools Fusion Features

2.1. Events

Events, from a Drools perspective are just a special type of fact, specifically, facts that record of a significant change of state in the application domain. While all events are facts, not all facts are events. In the next few sections the guide describes the specific differences that characterize events from other facts.

2.1.1. Event Semantics

An *event* is a fact that has thes distinguishing characteristics:

- **Usually immutable:** Since events are a record of a state change in the application domain, in other words, a record of something that already happened, and the past can not be "changed", events are immutable. 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 usecases for rules is event data enrichment.

  Tip

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

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

- **Managed lifecycle:** Due to their immutable nature and 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, once 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 with 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. For example, averaging 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.

Tip

A simple way to understand these 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 value. This declares that the type is to be handled as a regular fact.
- event : This declares that the type is to be handled as an event.

For instance, the example below declares 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 assiging 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 with them. While most of the metadata values have defaults that are automatically assigned to each event when they are inserted into the working memory, it is possible to change the default on an event type basis, using the metadata tags listed below.

In the following examples, we will assume you have 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> )
```

@role 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 is assigned to the event at the time the event is inserted into the working memory. It's also possible for the event has the timestamp as one of its attributes. In these cases, you 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

As we discussed earlier in this guide, Drools supports both these 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. You can define 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. We'll discuss the details on the effects of using this tag in the Memory Management section of this guide. 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. It is also 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, you would write:

Example 2.7. declaring the expiration offset for the VoiceCall events

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

2.2. Session Clock

Reasoning over time requires a reference clock. To illustrate one example, if a rule reasons over the average price of a given stock over the last 60 minutes, how does the engine know what stock price changes happened over those 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 the **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 a set of different scenarios that may require different clock implementations. Soem examples are:

- **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:** Uually, when running rules in production, an application will require a real time clock that allows the rules engine to react immediately to time progression.
- **Special environments:** Specific environments may have specific requirements on time control. For example, clustered environments may require clock synchronization through heart beats, or JEE environments may require the use of an AppServer provided clock.
- **Rules replay or simulation:** To replay or simulate scenarios it is necessary that the application controls the flow of time.

2.2.1. Available Clock Implementations

Drools 5 provides two clock implementations out of the box. The default is the real time clock, which is based on the system clock. There is also an optional pseudo clock, which is controlled by the application.

2.2.1.1. Real Time Clock

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

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

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

2.2.1.2. Pseudo Clock

Drools also offers a clock implementation that is controlled by the application. This implememtation is that is called "Pseudo Clock". This clock is specially useful for unit testing temporal rules since it can be controlled by the application, making the results more deterministic.

To configure the pseudo session clock, you set the session configuration parameter to "pseudo":

```
KnowledgeSessionConfiguration config = 
 KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
```

```
config.setOption( ClockTypeOption.get("pseudo") );
```

An example of how to control the pseudo session clock is:

```
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 deal with streams of events. These streams can be provided to the application in various forms, such as JMS queues, flat text files, database tables, raw sockets, or even through web service calls. Regardless of the form they may take, 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 (containing a single type of events), or heterogeneous, (containing 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 through which facts come. The facts may be regular facts or special facts such as events.

In Drools, facts from one or more entry point (stream) or event may join with facts from the working memory. They never mix in that they never lose the reference to the entry point through which they entered the engine. This is important to remember because while you may have the same type of facts coming into the engine through several entry points, but a fact that is inserted into the engine through entry point A will never match a pattern from a entry point B, etc.

2.3.1. Declaring and Using Entry Points

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

For example, in a banking application, transactions are fed into the system coming from streams. One of the streams contains all the transactions executed in ATM machines. 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 this 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 this example, the rule also joins the event from the stream with a fact from the main working memory (CheckingAccount).

Now, lets look at 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
```

This second rule will match events of the exact same type as the first rule (WithdrawRequest), but from two different streams. 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".

Entry points, therefore, 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 errors.

Inserting events into an entry point is equally simple. Instead of inserting events directly into the working memory, you 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 );
```

This example shows how to manually insert facts into a given entry point. Applications will usually use one of the many adapters to plug a stream end point, like a JMS queue, directly into the engine entry point without having to code 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 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. Hundreds of papers and thesis have been written and approaches have been described for several applications. Drools takes a pragmatic and simple approach based on several sources, including these:

[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.

Note that 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).

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. The operators are:

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

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.

For example:

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

This 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 (such as 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 at positive infinity.
- If no value is defined, it is assumed that the initial value is 1ms and the final value is positive infinity.

Tip

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. For 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 )
```

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 event belongs to the distance range declared for the operator.

For example:

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

This 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 (such as 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 at positive infinity.
- If no value is defined, it is assumed that the initial value is 1ms and the final value is at positive infinity.

Tip

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

```
$eventA : EventA( this before[ -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. For 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 )
```

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.

For example:

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

This 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 paratemer 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 )
```

This pattern will match if and only if:

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

Caution

It makes no sense to use negative interval values for the parameters for the coincides evaluator. The engine will raise an error if negative values are used.

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.

For example:

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

This pattern will match if and only if $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:

- If one value is defined, this will be used as the maximum distance between the start timestamp of both events and the maximum distance between the end timestamp of both events in order for the operator to match. For 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. For 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. For 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 $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
```

Caution

It makes no sense to use a negative interval value for the parameter. The engine will raise an exception if a negative value is used.

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 $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
```

Caution

It makes no sense to use a negative interval value for the parameter. The engine will raise an exception if a negative value is used.

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 $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 for the operator to 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.

For example:

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

This pattern will match if and only if $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 the 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
```

Caution

It makes no sense to use a negative interval value for the parameter. The engine will raise an exception if a negative value is used.

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.

For example:

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

This pattern will match if and only if $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. For example:

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

Will match if and only if:

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

Caution

It makes no sense to use a negative interval value for the parameter. The engine will raise an exception if a negative value is used.

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.

For example:

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

This pattern will match if and only if:

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

The overlaps operator accepts 1 or 2 optional parameters:

- 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. For 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.

For example:

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

This pattern will match if and only if:

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

The overlappedby operator accepts 1 or 2 optional parameters:

- 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. For 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. For 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.

For example:

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

This pattern will match if and only if $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. For example:

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

Will match if and only if:

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

Caution

It makes no sense to use a negative interval value for the parameter. The engine will raise an exception if a negative value is used.

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 )
```

This 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. For example:

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

Will match if and only if:

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

Caution

It makes no sense to use a negative interval value for the parameter. The engine will raise an exception if a negative value is used.

2.5. Event Processing Modes

Rules etngines in general have a well defined method of processing data and rules and providing the application with the results. There are not many requirements on how facts should be presented to the rules engine, because in general, the processing itself is time independent. While is may a good assumption for most scenarios, it is not universally true. When the requirements include the processing of real time or near real time events, then time becomes an important variable of the reasoning process.

The following sections explain the impact of time on rules reasoning in the context of the two modes (Cloud and Stream) 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, regardless of whether they are regular facts or events, as a whole. There is no notion of the flow of time, although events have a timestamp. 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, then activate and the fire rules.

This mode does not impose any kind of additional requirements on facts. For example:

- There is no notion of time. No requirement relate to clock synchronization.
- There is no requirement on event ordering. The engine looks at the events as an unnordered cloud against which the engine tries to match rules.

On the other hand, since there are no time-based requirements, some benefits of Fusion are also not available. For example, in CLOUD mode, it is not possible to use sliding windows, because sliding windows are based on the concept of "now" and, as we stated earlier, there is no concept of "now" in CLOUD mode.

Since there is no ordering requirement for events, it is not possible for the engine to determine when events no longer match and as so, there is no automatic life-cycle management for events. In other words, 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, it is possible to change this behavior either by setting a system property, using configuration property files or by using the API. The property to be set is:

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

The equivalent property is:

```
drools.eventProcessingMode = cloud
```

2.5.2. Stream Mode

In contrast to CLOUD mode, the STREAM processing mode is used when the application needs to process streams of events. STREAM mode adds some usage requirements, but it enables features make stream event processing possible.

The additional requirements to use STREAM mode are:

- Events in each stream must be time-ordered. 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. While the application does not need to enforce time ordering between streams, the use of non-time-synchronized streams may result in some unexpected results.

When these requirements are met, the application may enable the STREAM mode using the following API call:

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

Or, by setting the equivalent property:

```
drools.eventProcessingMode = stream
```

When using STREAM mode, the engine understands the concepts of flow of time and "now." In other words, the engine understands how old events are based on the current timestamp as 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

These features are explained in the following sections:

2.5.2.1. Sliding Window Support

In STREAM mode, the engine can perform calculations on moving (or "sliding") windows of interest, either temporal or length-based. Support for Sliding Windows, provides out of the box aggregation functions and leverages the Drools plugable function framework to allow for the use of users defined custom functions.

2.5.2.2. Automatic Event Lifecycle Management (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.

In contrast, in STREAM mode, the session clock is responsible for keeping the current timestamp. Based on the session clock, the engine performs all the temporal calculations on events aging, synchronizes streams from multiple sources, and schedules future tasks.

See secion ### in this guide for more details on the Session Clock section such as how to configure and use different session clock implementations.

2.5.2.3. Automatic Rule Delaying (Negative Patterns in Stream Mode)

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

When running in STREAM mode, however, 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 such a way that you do not need to perform any additional steps 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 previous rule will have the same effect as the following fule, but it does not give you the additional tasks of having 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
```

2.6. Sliding Windows

Sliding Window is an approach to scope events of interest as a the ones belonging to a window of time that is constantly moving. The two most common sliding window implementations are time based windows and length based windows.

The next sections in this guide describe these implementaitons.

Important

Sliding Windows are only available when running the engine in STREAM mode. Refer to the Event Processing Mode section in this guide for details on STREAM mode.

2.6.1. Sliding Time Windows

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

For instance, if you want 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 you want 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 2.14. 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
```

Note that the engine will automatically discard any SensorReading older than 10 minutes and keep the calculated average consistent.

2.6.2. Sliding Length Windows

Sliding Length Windows works in a similar way as Sliding Time Windows, but it discards events based on the arrival of new events instead of based on the flow of time.

For instance, if you want to consider only the last 10 XYZ company Stock Ticks, independent of how old they are, the pattern would look like this:

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

The pattern is similar to that used in the previous section with Sliding Time Windows, 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 you want 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 2.15. 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
```

Note that using this rule, the engine will keep only the last 100 readings.

2.7. Knowledgebase Partitioning

Warning

This is an experimental feature, and is subject to changes in the future.

The classic Rete algorithm is usually executed using a single thread. Although, as confirmed in by Dr. Forgy in his development of Rete, the algorithm itself is parallelizable. The Drools implementation of the ReteOO algorithm (RETE tailored for object-oriented systems) supports coarse grained parallelization through rulebase partitioning.

When this option is enabled, the rulebase will be partitioned into several independent partitions. 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 that multiple partitions may be "active" at a single point in time.

Everything involving knowledgebase partitioning should be transparent, except that all working memory actions (i.e., insert/retract/modify) are executed assynchronously.

Important

This feature enables parallel LHS evaluation, but does not change the behavior of rule firing. 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, you should 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? (for example, do the rules 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, you must set the following option:

Example 2.16. 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 you 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 2.17. 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 in this guide for additional details on STREAM MODE.

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 2 approaches 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 approach to allow the engine to calculate the window of interest for a given event type is by explicitly setting the expiration offset. To do this, you use the declare statement and define an expiration for the fact type:

Example 2.18. 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 approach 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 2.19. 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
```

In this example, when analyzing the 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. The implicit expiration offset for BuyOrderEvent is therefore 10 seconds. AckEvent, on the other hand, can only match existing BuyOrderEvents, 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

Index