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 begginings, 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 refered as CEP systems.

This is a very simplistic explanation to a really complex and fertile field of research, but sets a very highlevel 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 infering 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 seemless 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 importante 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 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.
- 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 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 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 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, 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.

Tip

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 bellow 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 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 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 bellow.

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
```

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 array of different scenarios that may require different clock implementations. Just to mention a few:

- **Rules testing:** testing always requires a controled 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.

Chapter 3. References

Index