Drools RETEOO (for dummies): #1 Intro

Jun 5, 2011
by Mauricio Salatino

Once we understand the basics of the RETE algorithm we need to go deep and analyze how the RETE network is built in order to optimize the rules that you write.
For that reason the following slides will show very simple examples and their generated Networks in order to play a little bit with the algorithm. We will analyze how the Network is built and also how it behaves at runtime when facts are asserted and they need to be matched with the rules.

First Example

As we already saw, the RETE network is built using all the Left Hand Side of our rules.
If we start with a simple rule with a single pattern like:

```
rule "Match all the persons with name Salaboy"
  when
    $person: Person(name == "Salaboy")
  then
    …
end
```

We will get the following Rete Network:

Basic
RETE

This simple network contains the following nodes:
A Root (or Rete) node that will serve as first entry point for our Facts. This node is followed by one Entry Point Node, in this case the DEFAULT entry point. If we start using Drools Fusion and define more named entry-points our network will have different points of entry for our facts and for our different event sources.

The Entry Point Nodes are always followed by Object Type Nodes. This nodes will specify the type of object that they are filtering. Each Object Type node can only filter one type of Object. If we analyze this node properties, we will see that the object type specified in this case is:

Object Type = [ClassObjectType.wordpress.salaboy.drltest.Person]

After the Object Type nodes, the alpha network begins. Inside it we will find what we call Alpha Nodes. This alpha nodes are simple filters over the object (fact) fields.

In this case, because we have a very simple rule the next node is a Left Input Adapter node, that transform a single object that was propagated from the alpha node to a Tuple that contains this single object. We need to do this in order to reach the Terminal node that generates the activation of this rule.

In this case the Right Hand Side of the rule will receive the Tuple (in this case with a single object). The RHS will be able to use the Tuple to execute some actions. Inside the right hand side we can access to the information contained in the Tuple and we define variables (the $person in this case) we will be able to retrieve it using the variable name.

If you take a look at the properties of the Terminal Node, you will see that it contains a property called Rule, that specifies the rule that was activated.

Rule = Match all the persons with name Salaboy

This information is used to locate the compiled version of the RHS of the rule when the rules its fired.

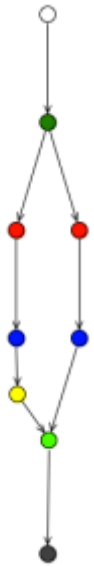
Let's move to an example that use multiple Patterns to evaluate a situation.

Multiple Patterns

Now we have the following rule:

```
rule "Match Person and Address"
  when
    $person: Person(name == "Salaboy")
    $address: Address(addressLine1 == "nowhere")
  then
    System.out.println("Hey I just find " + $person + " and an Address: "
+ $address );
end
```

Now the RETE network looks like:



Multiple Patterns

WARNING: Notice that this network only represent the rule described before and not multiple rules. In other words, this network represent the rule for this example and not all the rules for previous examples.

As you can see, we have a similar structure that the one that we got in the previous example. But now, we are evaluating two different patterns: Person and Address. Because the DRL language use an implicit AND between pattern definitions, the previous rule represent:

Person(...) AND Address(...)

For that reason a new node type its introduced inside our network. We need a way to join both object types to identify when we have a Person AND an Address.

The green node inside our network is a Join Node. This node is inside what we called Beta Network. The Beta Network its responsible for all joins between different patterns. Inside the Beta Network we will have nodes

that will be in charge of having constraints and the logic to mix information from multiple patterns. All the Beta Nodes have Two inputs (Left and Right) and One Output.
The Left Input from a Beta Node receive a Tuple of values that will be compared with the Right Input that receive just one value. Beta Nodes has memories to store previously propagated Objects that are probably waiting for more information to be able to continue its propagation. We will see more about this in the runtime evaluation.

Notice that in this case, because the Person() pattern is the first one in our rule, after the alpha node that checks for the name, the object needs to be propagated to a Left Input Adapter before going to the Left Input of the Join Node. Just for you to know if you invert the order of the Patterns in the rule, the Address Object will be propagated to the Left Input adapter instead of the Person Object. It's just a minor change.

Constraints inside Join Nodes

If we change the previous rule to the following one:

```
rule "Match Person and his/her Address"

when
$address: Address(addressLine1 == "nowhere")
$person: Person(name == "Salaboy", address == $address)

then
System.out.println("Hey I just find " + $person + "that lives in: " +
$address );
 end
```

We will get a very similar network with the only difference that the Join Node, now will contain a constraint inside it.
If you inspect the properties of the Green Node in our network, now you will see an extra property:

Constraint 1 = [ClassFieldExtractor.wordpress.salaboy.drltest.Person field=address] Object == (ValueType = 'Object') $address

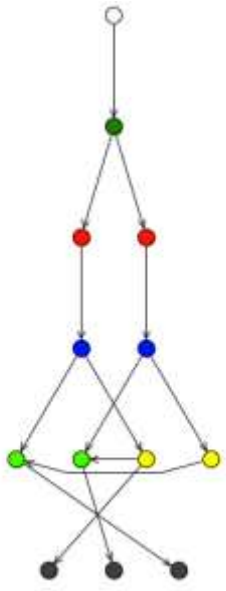
Notice that we are filtering the objects in the join using the Person.address value and the $address variable that is bound to an Address pattern. If the reference of the Person.address object is the same that the object that was bound to the $address variable, this join will evaluates to true and it will propagate this specific Person and Address object, in this case to the terminal node.

For making this rule activated we need to insert into the working memory the Person object, the Address object and also we need to make sure that the Person.address field is pointing to the Address object that we have inserted.

```
Person salaboy = new Person("Salaboy", 28);
 Address address = new Address("nowhere","",1425,"BA");
 salaboy.setAddress(address);
 ksession.insert(salaboy);
 ksession.insert(address);
```

Node Sharing

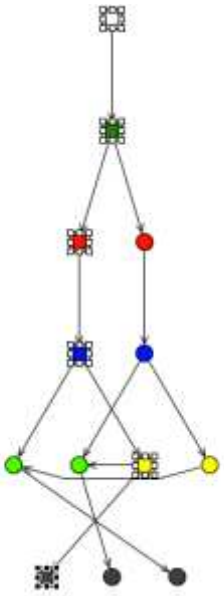
Now, What happen if we include the rules from the previous three examples in the same DRL file? Now our network will represent all the rule set and not just only one rule. To improve performance, when we add multiple rules the RETE algorithm will try to share nodes that are being used for more than one rule.



Node Sharing

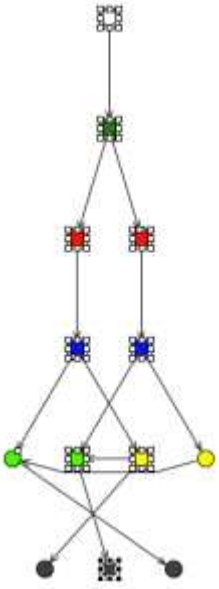
Starting from the bottom of the Network, we can see that we have One terminal node per Rule. Because our rules just filter Persons and Addresses we have just two Object Type Nodes (OTN). Inside our alpha network we just have two alpha nodes that are reused by all the rules. In this case this alpha nodes are being shared among a set of rules that are using them.

When we want to analyze where is the first rule represented, we discover that the Left Input Adapter nodes are also being shared. The first rule its composed by the following selected nodes:



First Rule

Notice that the Left Input Adapter (yellow node) is being shared with the second rule. The second rule nodes are selected in the following figure:



Second Rule

For finding the right nodes of the second rule, you need to inspect the Join Nodes, to see what is the one that doesn't contain a constraint. You can deduce the third rule based on that as well.

Runtime Analysis

In this section we will be analyzing the runtime behavior and the drools internal implementation in order to understand the basics.

In order to run all the previously described rules you can download the code here and run the tests provided. Different tests run each rule separately and all the rules together. You can see and inspect the information that it is being inserted into the working memory for the rules to fire.

Once we understand which rules are activated we can start analyzing the internals. For being able to

understand how the engine is working in the back we need to know a couple of implementation details.

Let's start with the basics inside the org.drools.reteoo.* package. All the nodes from the Alpha and the Beta Network are described there.

The Rete class contains the definition of the first node in our Rete network. You can see that this Rete class extends the ObjectSource class and implements the ObjectSink interface.
The ObjectSource class implements the logic that its necessary to keep the reference from where a fact handle was propagated to the current node.

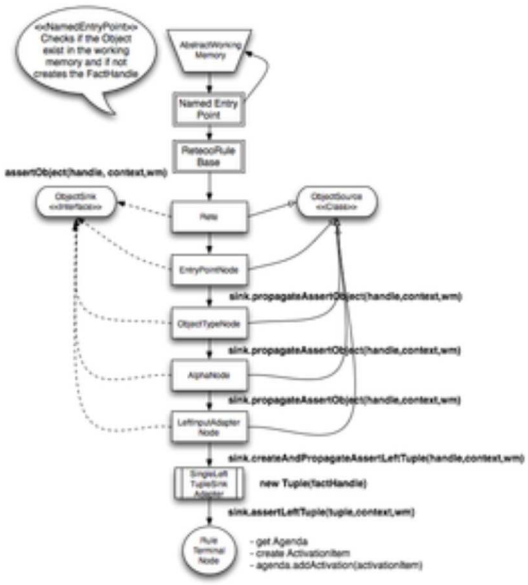
The ObjectSink interface expose the methods that needs to be implemented by a Node that can receive propagated FactHandles.

The following methods are defined in the ObjectSink interface:

```
 public void assertObject(InternalFactHandle factHandle,
 PropagationContext propagationContext,
 InternalWorkingMemory workingMemory);

public void modifyObject(InternalFactHandle factHandle,
 ModifyPreviousTuples modifyPreviousTuples,
 PropagationContext context,
 InternalWorkingMemory workingMemory);
```

If we debug the first test, that only contain one rule and it evaluates just one single pattern we will find that:



Simple Implementation

Remember that we are inserting a Fact that generate the rule activation.

1. AbstractWorkingMemory will assert the fact in the default entry point inside the AbstractWorkingMemory.insert(...) method.
2. The EntryPoint Node will take a look at its cached ObjectTypeNode[] nodes, in this case just one that represent the OTN Person. And for each cached Node it will call the assertObject(...) method. Notice

that here we can deduce that the only possible nodes after an Entry Point node are OTN.

3. The ObjectTypeNode will receive the call to its assertObject that will propagate the fact handle if and only if the object is for that certain type, in this case Person.
   Notice that the OTN has memory and it store the object inside this memory if the ObjectMemory is enabled. The OTN at this point will look for it sink node where it can propagate the FactHandle. In this case the Sink node is the alpha node.
4. The alpha node will check its internal constraint and if it validates to true it will get its sink and it will propagate the fact handle.
5. The sink for the AlphaNode in this case its the leftInputAdapter that it is in charge of using a SingleLeftTupleSinkAdapter to create a new left tuple that is required by the RuleTerminalNode
6. The RuleTerminalNode will get the agenda, create an activation item with the LeftTuple that contains the factHandle with the matched object and add the new ActivationItem to the agenda.

Sum up and Links

During this post I've just defined the basic structure and terms that we need to understand in order to analyze how our rules are being handled in the back by the RETE algorithm. Understanding how the network behaves will help us to optimize our rules and the overall performance of our network. In following posts we will analyze how the Beta network behaves using more complex rules.

You can download and play with the examples from here: https://github.com/Salaboy/Drools_jBPM5-Training-Examples

Inside that repository you can find the examples described in this posts here: **Drools_jBPM5-Training-Examples** / drools5 / 09-DroolsExpertRETEnetworkExamples

If you have Eclipse and the Drools/jBPM5 plugins available you will be able to run ***mvn eclipse:eclipse*** and then open and inspect the provided DRL files.

Some Notes:

If you are debugging insertions, remember that on initialization the InitialFact its inserted before your facts.

Filed under: drools , English , Java , JBoss , JBoss Drools , JBoss JBPM , jbpm5 , PlugTree Tagged: algorithm , brms , cool , Drools , example , forgy , jboss rules , matching , network , pattern , rete , rete network , reteOO , rules , simple Comments: 2 -------

algorithm _ BRMS _ cool _ Drools _ english _ example _ forgy _ Java _ JBoss _ jboss drools _ JBoss JBPM JBoss Rules _ jBPM5 _ matching _ network _ Pattern _ PlugTree _ Rete _ rete network _ reteOO _ rules simple

Original Post