JBoss Rules

A High Level Analysis on “Else” and “Otherwise”

Edson Tirelli

The intent of this document is to provide a technical analysis on “ELSE” and “OTHERWISE” features, covering semantics and possible implementations.

It is not the intent of this document to discuss if they are good or bad features. So, besides being good or bad, if we decide to implement, this is my technical opinion.

“Else” is in fact a specific case of “otherwise”, as we can think of “else” as an “otherwise” for a single rule. So, I will first analyze “else” being the specific scenario and then generalize the analysis for otherwise.

Throughout this document I will use some rules as examples. The syntax used to write these rules must be read as pseudo-code, as it is not the intent of this document to define any syntax.

1. ELSE

1.1. Semantics:

The first important decision we need to make is about semantics. Let’s see one example:

```
Rule “else sample”
when
    $p : Person( name == ‘Bob’, $likes : likes )
    $c : Cheese( type == likes )
then
    // (1) do something
else
    // (2) do something else
end
```

I see two possible semantics for “else” in the above example.

a) The first possible semantic is to interpret the else as the RULE else. It means if the rule is not activated, the block (2) gets executed. It means, block (2) will not be executed (if the rule fires at least once) or will be executed only once if the rule is never fired.

b) Second possible semantic is to interpret the else as the PATTERN else. It means that for each tuple [Person, Cheese], the rule will either fire block (1) if the constraints are successfully matched or block (2) if any of the constraints fail.

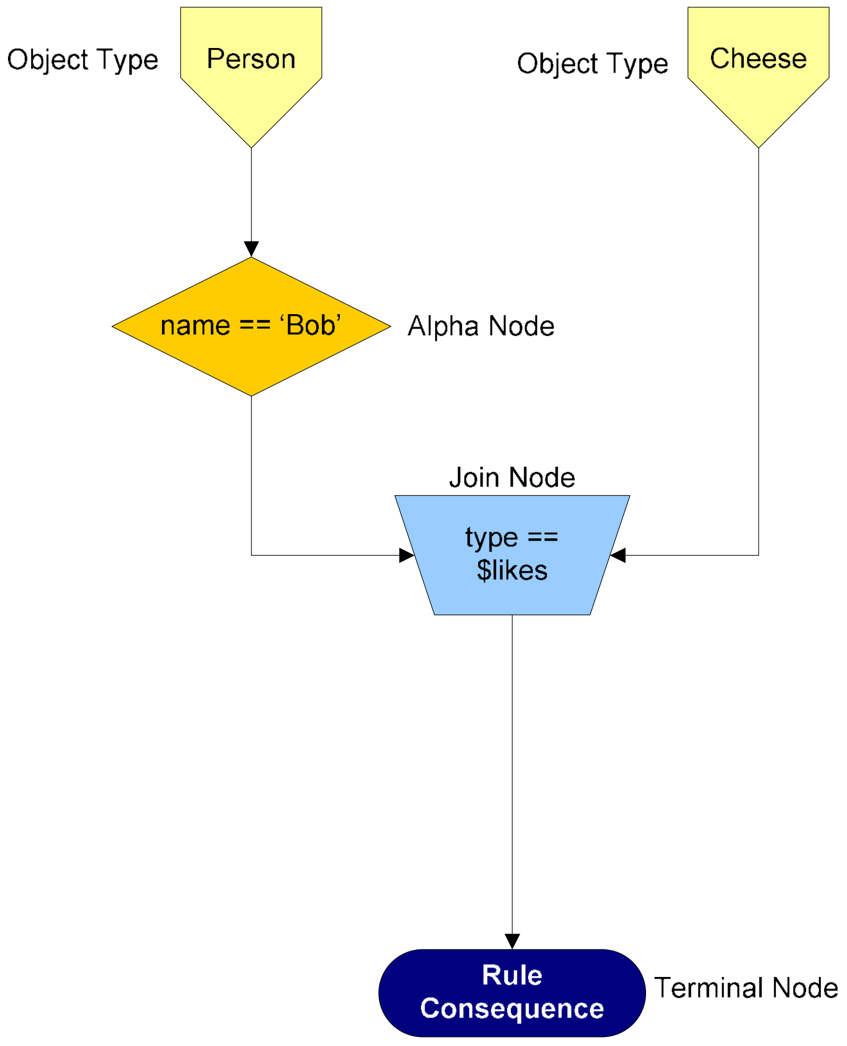
Semantics (a) above is not useful IMO because the actions you can execute in the block (2) would not be able to rely on (or use) facts, as there is no bound fact. Also, you can’t know in such a construction if the rule did not executed because the patterns failed to match or because there wasn't any fact in the rule base that matched the required object type. An else like that would only be useful I think in "setup rules", that will basically assert new facts into the rule base, what is very limited I think.

Semantics (b) above is the general case IMO, as you can execute actions over bound facts. It is almost like writing a new rule negating all constraints.

OPSJ uses yet another semantics that consists of tagging conditions and adding else clauses for each tag (http://www.pst.com/opsjbro.htm). In my limited experience, I see it more as a complication than as a help, but if you think it is a good solution, please share your thoughts with me.

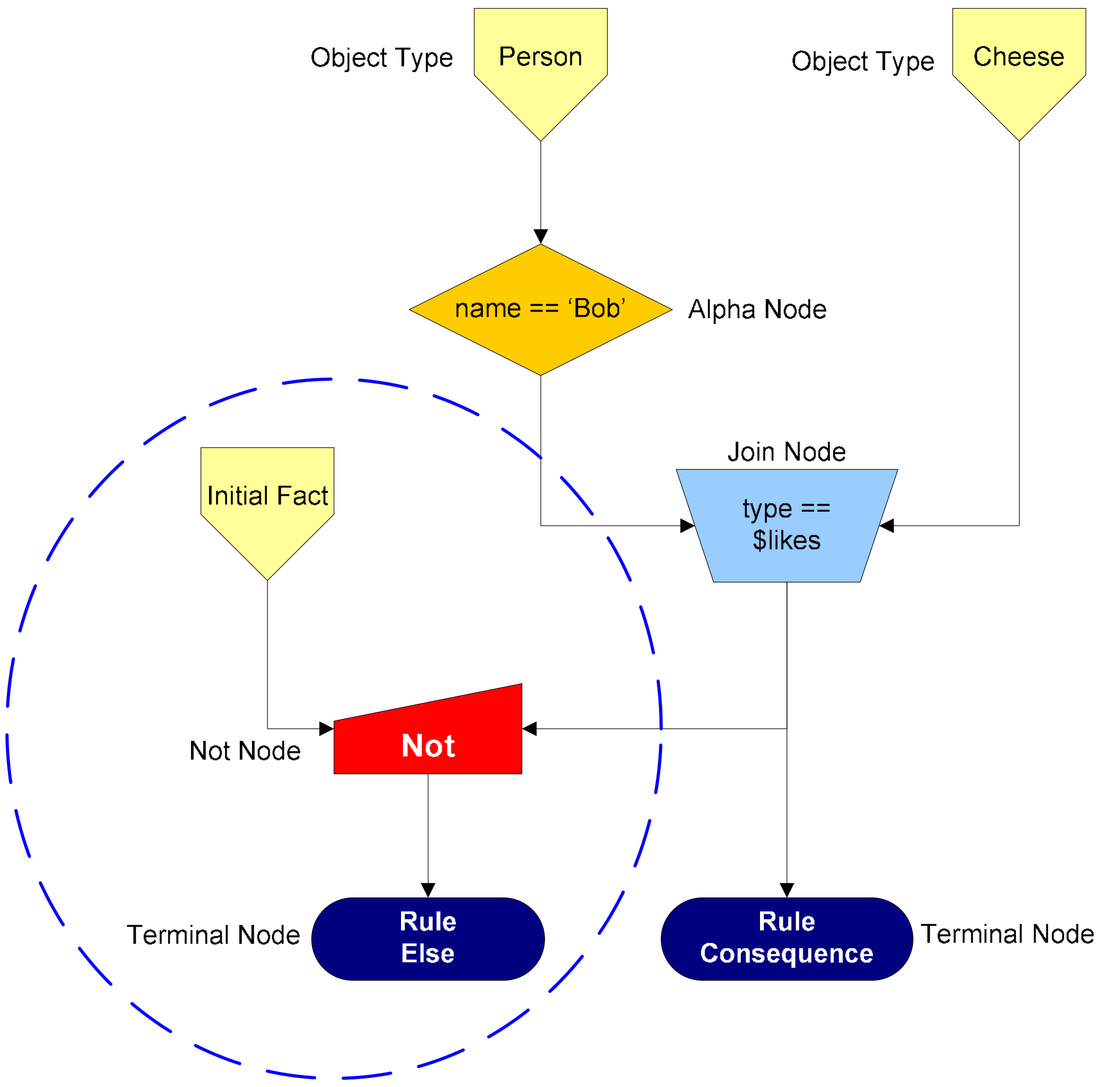
1.2. Implementation:

From an implementation perspective, what I would like to suggest is that we use the usual Rete nodes to implement whatever semantics we want to. So, the basic network for the above rule looks like (not showing adapter nodes):



[Figure 1: A simple rule network]

Adding support to “else” would be simply to add nodes to the network to provide the needed functionality. So, implementing else with the (a) semantics discussed before would be to add the following to the network:



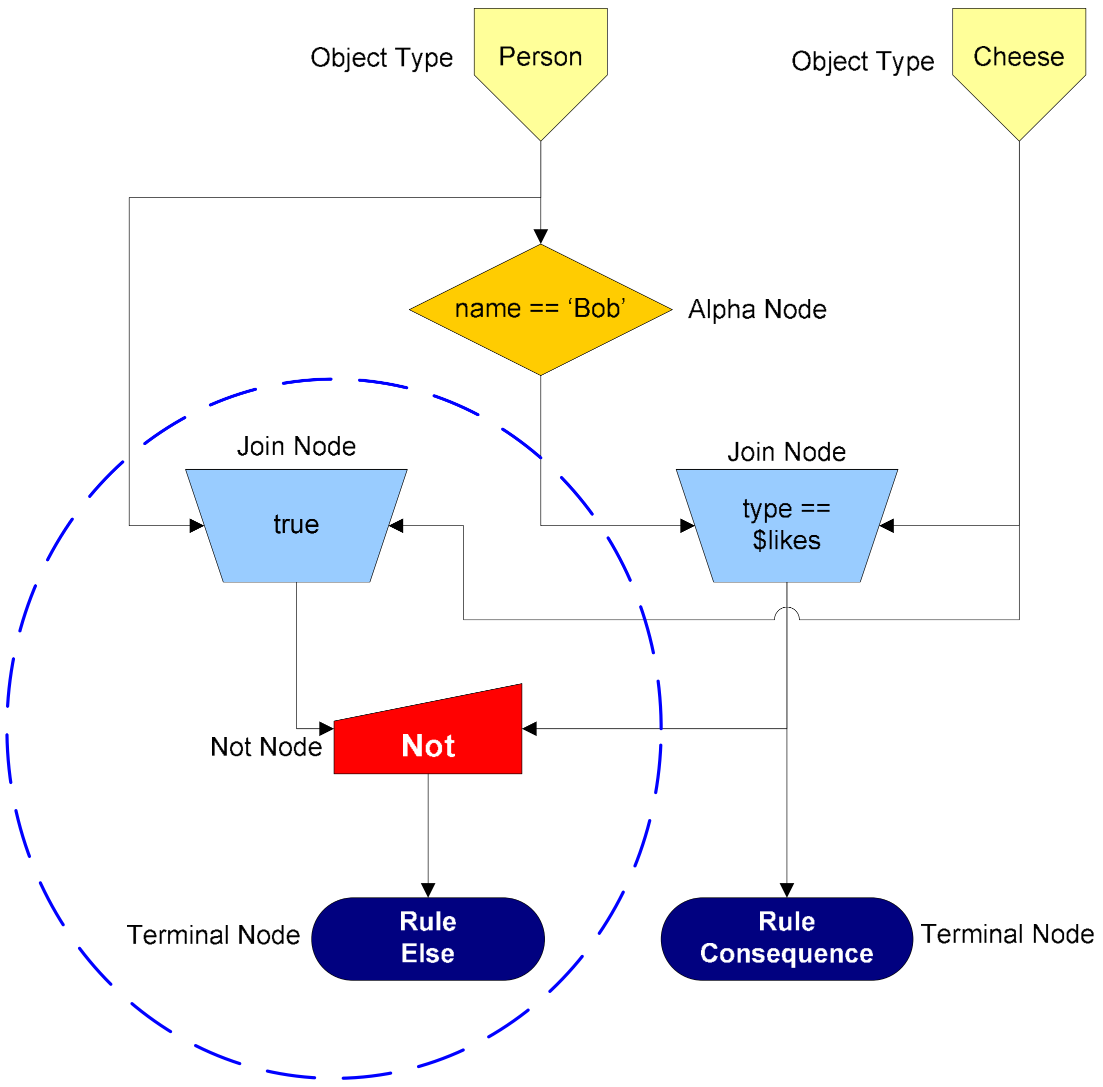
[Figure 2: Adding “else” with semantics (a)]

In the blue circle you find the nodes added to provide the else functionality. Basically a NOT node, left activated by an InitialFact and right activated (RightAdapterNode not showed in the draw to make the draw cleaner) by the propagations of the last node before the TerminalNode for the rule.

So, in this case, if the rule is activated by any tuple, it will cause the NOT node to deactivate the “else” TerminalNode.

This, IMO, is a very clean way to implement the “else” functionality. It is easy to maintain as it is basically restricted to the parser and the builder, and it is consistent with the current framework, automatically leveraging all current core features like the truth maintenance system (logical assertions), IDE support and Rete Viewer, etc.

The implementation of the semantics (b) discussed before is almost the same, but instead of the NOT node being left activated by the InitialFact, it is left activated by the same tuples that may eventually activate the rule. See the network bellow:



[Figure 3: adding "else" with semantics (b)]

As you can see, the idea is exactly the same as the previous one. All we did was to add join nodes without constraints from the ObjectTypeNodes propagations to create the tuples that will left activate the NOT node. This will also automatically provide for variable references to be used in the "else" block. The NOT node uses a single TupleEqualsConstraint constraint that matches the tuple from the left with the same tuple arriving from the right.

So, every tuple will either activate the Rule Consequence or the Rule Else block.

2. OTHERWISE

As previously stated, “otherwise” is the generalization of “else” for multiple rules. The same way that we did for “else”, we need to choose between the different possible semantics for “otherwise”. So will we trigger “otherwise” rule only once (a) or will we trigger it for each tuple that didn’t matched any of the stated patterns (b)?

It is easier to visualize it in a decision table:

	Person	Cheese	Consequence
Rule 1	$likes:likes	Type == $likes, Price < 20	// buy 10 pieces
Rule 2	$likes:likes	Type == $likes, Price < 50 & >= 20	// buy 5 pieces
Otherwise			// buy 1 piece

Or in a more “real world” telecom guiding use case:

	Usage Event	Pricing Plan	Consequence
Calling Circle	$pp:pricingPlan	Code == $pp, Type == “CC”	// charge fixed rate
Reduced price	$pp:pricingPlan, callTime > ’20:00hs’	Code == $pp, Type != “CC”	// charge discounted price
<other rules>	…	…	…
Otherwise			// charge regular price

So, for the above cases, we need it to trigger for each tuple (semantics (b)), and that is what I think it is best, but again, I’m open for discussions.

2.1. Implementation

My proposed implementation for either of the discussed semantics is also just a generalization of the “else” proposed implementation. Instead of a single NOT node in the “else” network, the “otherwise” network would be a sequence of NOT nodes, being one for the conclusion of each rule. So, if we have 3 rules and we add an otherwise for it, the “otherwise” network will have 3 NOT nodes in sequence, negating each of the rules’ conclusions.

2.1.1. Open issues

My only unanswered question for the implementation is what happens if the users define different “columns” (patterns) for the rules in an “otherwise” group. This does not happen in decision tables as we have a tabular table with the same pattern (but possibly different constraints) for each rule.

In DRL form though, a user can use patterns A and B for the first rule and patterns A, C and D for the second rule. So, if we use semantics (b) discussed before, what would be the tuple activating “otherwise”? [A, B, C, D]? I think we may have unexpected behavior there. So, for DRL form, we may want to restrict either by syntax or by build time analysis, the correct formation of the rules in each “otherwise” group in a way that all rules have the same patterns (I’m not talking about constraints). For Decision Tables, I don’t think it is a problem as tables are already written with the same patterns for each rule.