Procedural logic in the reasoning process

> *This column is the next in a series that provides the reader with best practices on using or choosing a rules engine. The target audience for this series is typically the user of a rule engine, i.e., a programmer or someone with programming skills. All coding examples should be read as pseudo-codeand should be easily translated to a specific target syntax for a rule engine that supports backward and forward chaining in an object-oriented environment.*

In this column I describe a general approach for using procedural logic during the inference processing. I assume that you already agree with the fact that it is not a good idea to have this kind of processing within the rules (see my last column[1]). Therefore you will understand that we have to come up with an approach that enables us to implement the desired logic independent from the rules!

The following paragraphs describe how to use procedural logic during the chaining process rather than writing procedures just before or after the infer block.

When needed logic

This paragraph deals with the need for some additional processing whenever data is needed. In bad examples for rule-based systems you may find rules such as these:

example rule 'user dialogs in rules'

```
ifrule isunknown(age)
then
  userdialog.askquestion(age, "what is
the age of the applicant?")
end
```

example rule 'database access in rules'

```
then
   applicatantdata.queryapplicantage(id)
end
```

example rule ‘chaining in rules’

```
ifrule isunknown(age)
then
    infer
     agerules( )
   backwardchain(age)
  end
end
```

In the previous paragraph I stated that these are bad examples. I could go on for hours but there’s one important thing that must be clear to you: These kinds of rules are only useful (if ever) in a ***backward*** chaining strategy because in a forward chaining strategy you could just as well retrieve the value for ‘age’ before you started the inference process. My point is: In a forward chaining mode you want to execute rules based on the values of attributes in premises of rules, thus their value has to be available. In that case you can just as well code the rules in a traditional manner before you enter the inference context.

Ok, I assume that you agree with the fact that these rules may only be necessary in a backward chaining strategy. I’m now going to describe a technique that enables the developer to achieve the same thing without cluttering up the rules with GUI, Data Transactions, or nested inference statements.

I’d like you to consider the following question for a moment: When does an inference engine in backward chaining mode evaluate either of the example rules? Answer: When the ‘age’ attribute becomes a (sub)goal. A more technical term is: The attribute age is ‘sourced’ when it is needed in the chaining process.

Now, some inference engines offer monitoring methods that will result in an event when the engine needs to evaluate an attribute. Let’s call this event-method ‘WhenSourced’. The method will be called when a monitored attribute is ‘sourced’ by the rule engine. In this call, a reference to the sourced attribute is passed along with an indication (lastchance) whether the inference engine has noticed that there are also rules that *may* be able to assign a value to the attribute.

So if ‘lastchance’ is true there are no rules that can solve the value of the attribute or these rules have already been tried.

Here’s how the first example mentioned above would be implemented using the WhenSourced method:

example code infer block

```
infer
applicationrules
enginelinkgoal(_engine.enginegetcurrent(),
->age)
//enable the WhenSourced event
backwardchain(->applicant_accepted)
end
```

The attribute ‘age’ is explicitly linked to the inference engine instance associated with this infer block. Now, if somewhere in the ‘ApplicationRules’ the ‘age’ attribute is referred to while its value is unknown, the WhenSourced method is sent to the Applicant class (the parent class of the attribute).

example code specialized whensourced method in class applicant

```
input pat is attributepointer
input lastchance is boolean
if pat = ->age
then
if lastchance
then
userdialog.askquestion(current.age,
“what is the age of the applicant?”)
else
// there are rules that may derive the
value of age
end
elseif pat = …
then
…
end
```

Notice that:

- The other examples can be done in a similar way.
- The implementation of the WhenSourced method could have been done in a generic way by using meta-programming features. In that case you don't have to write extra code if you introduce a new attribute.
- Normally you don't want to bother with identifying which attributes should be linked to the inference engine. In that case you can call the following example method (that you would create in the super class of your domain objects):

example code linkAllAttributes

```
var eng, attr_idx is integer
var pat is attributepointer
eng = enginegetcurrent
attr_idx = 1
loop
pat = getattribute(current, attr_idx)
breakif pat = null
enginelinkgoal(eng, pat)
end
```

The EngineLinkGoal method links an attribute to the *current* inference engine instance. This means that if the attribute is sourced in a nested infer block or, more generally, in another infer block, the WhenSourced method is not sent.

Reactional logic

In the previous paragraph I discussed how you could implement procedural logic when a value of an attribute in a rule is needed. Now I will deal with two situations where you want to execute some procedural processing as soon a rule fires.

Using a monitor

You can react on a rule that fires by using a monitor. The example here is that you want to show why a rule fires with an explanation text to the user — immediately when this rule fires.

code snippet

```
ifrule current.age < 18
then
declined = true
giveexplanation(“applicant needs to
be at least 18 of age”)
end
```

I’m sure you can relate to this kind of rule. The problem is that the ‘GiveExplanation’ method will probably not be declarative (and again, it does not have a lot to do with the business logic either). So, let’s look at a solution that gets rid of the procedural logic in the action of the rule.

Some inference engines offer monitoring methods that will result in an event when the engine changes the value of an attribute. Let’s call the method ‘WhenModified’.

This solution creates a class called ExplanationMonitor.[2] Specialize the Create method of this class and code it as follows:

example code constructor method of explanation monitor

```
var p is pointer to currentclass
p = up.create(name)
p.attach(null, ->applicant)
// enable the WhenModified event
return(p)
```

The attach method above attaches the Applicant class to the ExplanationMonitor instance. From that moment on, the inference engine should send the WhenModified method to the ExplanationMonitor instance when there is a change to any attribute of any instance or to any class attribute of the Applicant class.

Here’s the code for the (specialized) WhenModified method of class ExplanationMonitor:

example code when modified

```
argument
input pat is attributepointer
if getattributename(pat) = “explanation”
then
giveexplanation(getstringvalue(pat))
end
```


In order to make this work you have to create an attribute Explanation in class Applicant. The example rule will look as follows:[3]

example rule

```
ifrule current.age < 18
then
declined = true
  explanation = “applicant needs to be
at least 18 of age”
end
```

Using the engine-history option

Next, I will explain how you can react on the result of the inference engine by using the history option. In this example you want to show which rules have fired when the inference process is finished.

In many applications, in addition to evaluating the rule, the developer also has to maintain the reasoning path in order to explain the solution to the (expert) user. A common way to do this is as follows:

code snippet

```
rule “applicant needs to be at least 18 of age”
ifrule current.age < 18
then
declined = true
logfiredrule(“applicant needs to be
at least 18 of age”)
end
```

Although LogFiredRule may be implemented in a declarative way, it has nothing to do with the business logic. Luckily for some inference engine users there is a history feature that enables you to retrieve all inferencing information after the backward chain statement. I will give an illustrative solution:

example code infer block

```
infer history
applicantrules
  backwardchain(->declined)
logrules
end
```

The LogRules method retrieves all the fired rules and presents them to the user.

example code logrules method

```
var eventhandle, rulehandle is integer
var fired_rules is list of string
for enginegethistory(enginegetcurrent()),
eventhandle
for eventgetrules(eventhandle),
rulehandle
add(fired_rules,
rulegetname(rulehandle))
end
end
showtouser(fired_rules)
```

This article was originally published by BRCommunity (link).

[1] Silvie Spreeuwenberg, “What about Methods in Rules?” *Business Rules Journal*, Vol. 9, No. 5 (May 2008), URL: http://www.BRCommunity.com/a2008/b418.html

[2] Note: You can use LibRT’s verification component *Valens* to find out which explanations subsume each other and which explanations exclude each other by running the ContradictoryChain check and the Redundancy check. This Contradiction check will list all the rules that assign a different explanation for the same condition, while the redundancy check will list all conditions that raise the same explanation

[3] Note: If you use constants instead of literal strings, you can easily see which explanation is generated by which rule

