Rule history and versioning (part 2)

> This column is one in a series that will provide the reader with best practices on using or choosing a rules engine. The following topics will be discussed in future columns:
> • magic values
> • rule templates
> • rule on/off
> • exception handling
> • what to do procedural, when to do rules
> • local variables in rules
> • arrays and chaining
> • forward chaining over multiple instances
> • backward or forward chaining?
> 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-code* and should be easily translated to a specific target syntax for a rule engine that supports backward and forward chaining in an object-oriented environment.

Today we continue our discussion on how to deal with rule versioning and rule history in a declarative way. In our last column[1] we described the characteristics of the rule history and versioning problem; we discussed an example and a simple solution. The disadvantage of the proposed solution is that the performance may decrease if a lot of rules are subject to rule versioning. The advantage of the proposed solution is that it is declarative and uses the rule engine as much as possible, which allows us to use automatic verification techniques.

In this column an alternative approach is discussed. We will discuss a solution to the rule versioning problem in which the advantages of the previous solution are (as much as possible) the same but the solution is more efficient.

From now on the term ‘rule versioning’ will be used to denote rule versioning and history.

Solving the rule versioning problem in an efficient way

The inefficiency in the first proposed solution can be apparent when you have a lot of historic

rules that are not applicable in a particular period due to rule versioning. The inference engine needs to evaluate in this situation a lot of rules that are not applicable. To solve this problem we will divide the task in two sub-tasks:

- First we are going to evaluate rules that post the rules that should be evaluated for a given task, situation, or event.
- Then we perform the task itself.

We will illustrate the approach with our previous example. The rules deriving the markdate do not change:

example code

```
rulesetmarkdate()ifrule markdate – normal case
if task = “calculate premium”
and event = “normal calculation”
then
markdate = currentdate
end
ifrule markdate – recalculation case
if task = “calculate premium”
and event = “re-calculation”
then
markdate = period.startdate
end
```

Based on the markdate we create rules that decide what rules to post:

example code

```
rulesetpostrules()ifrule 1 – version 1
if markdate >= 01–01–1999 and markdate < 01–01–2001
then postruleexamplerule1-v1
postgeneralrules
postrules = true
end
ifrule 1 – version 2
if markdate => 01-01-2001
then postruleexamplerule1-v2
postgeneralrules
postrules = true
end
```

PostRuleExampleRule1-v1, PostRuleExampleRule1-v2 and PostGeneralRules are rule sets. In this

example each rule set consists of one rule. PostRuleExampleRule1-v1 contains the rule ExampleRule1 version 1. PostRuleExampleRule1-v2 contains the rule ExampleRule1 version 2. PostGeneralRules contains the rule ExampleRule 2.

Preferably we group rules in rule sets that should be active in the same period. The Boolean attribute ‘PostRules’ is used as goal in the backward chain process. Since a backward chain process stops when the goal is derived this assumes that one rule concludes the exhaustive set of applicable rules! In the example this constraint forces us to post the ruleset “PostGeneralRules” in the conclusion of both rules.

When we perform the task we first evaluate the rules that post the applicable rules. We can do this with a backward chain on the attribute “PostRules” statement. After that we will perform the task (in our previous example in a backward chain mode).

example code

```
task()
var postedrules is list of string
start inference
rulesetmarkdate()
rulesetpostrules()
postedrules = enginegetrules(enginegetcurrent)
  backwardchain(->postrules)
disablerules(postedrules)
backwardchain(-premium)
end
```

The statements ‘PostedRules = EngineGetRules(EngineGetCurrent)’ and ‘DisableRules(PostedRules)’ are introduced to improve the efficiency in the second backward chaining process. The rules in the rule sets ‘RuleSetMarkDate’ and ‘RuleSetPostRules’ are no longer needed and can therefore be disabled.

In some rule engines you can write a method that disables rules while you continue using the same rule engine instance. If that is not possible you can restart your rule engine with the posted rules.

The method EngineGetRules provides you with all rules that are ‘known and enabled’ by the rule

engine. A handle to the active rule engine instance is given by the method EngineGetCurrent.

The method DisableRules can be implemented as follows:

example code

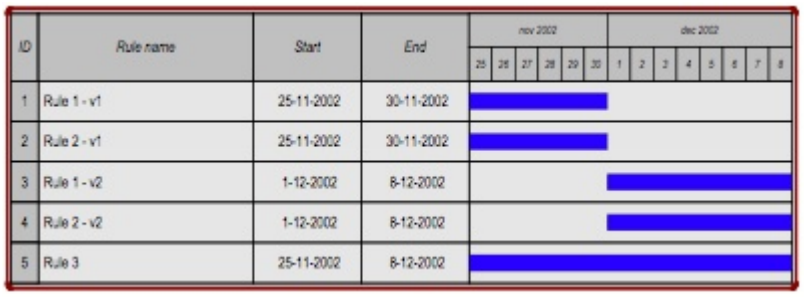
```
disablerules(in rules: list of integer)
var r is integer

for rules, r
engineruleenable(enginegetcurrent(), r, false)
end
```

The method EngineRuleEnable should be provided by your rule engine API.

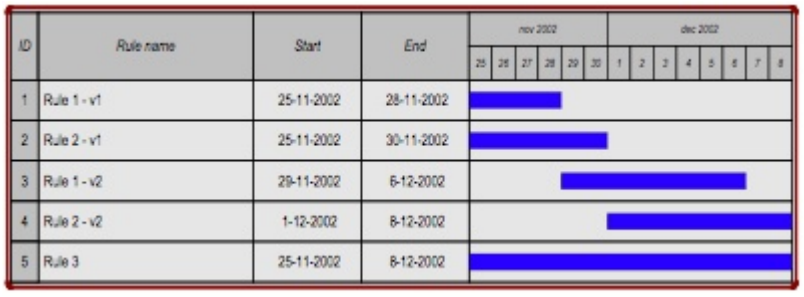
Evaluation solution

The disadvantage of this solution is in the maintainability of the rule set ‘RulesetPostRules’. This rule set will contain a lot of redundancy and a lot of rules when there is little overlap in the applicability period of rules. The disadvantage will be shown with two diagrams:

situation 1



situation 2



In situation 1 the RuleSetPostRules will consist of two rules:

example rule situation 1

```
ifrule rulesnovember
if markdate > 25–11–2002 and markdate < 01–12–2002
then postrulesversion1
postrule3
end
```

where PostRulesVersion1 is a ruleset containing rule 1 — v1 and rule 2 — v1.

example rule situation 1

```
ifrule rulesdecember
if markdate => 01–12–2002 and markdate < 08–12–2002
then postrulesversion2
postrule3
end
```

where PostRulesVersion1 is a ruleset containing rule 1 — v2 and rule 2 — v2.

In situation 2 the RuleSetPostRules will contain four rules, one for each rule end-date period:

example rules situation 2

```
if markdate > 25–11–2002 and markdate < 28–11–2002
then postrule1version1
postrule2version1
postrule3
endruleperiod2
if markdate => 30–11–2002 and markdate < 30–11–2002
then postrule1version2
postrule2version1
postrule3
end
ruleperiod3
if markdate => 28–11–2002 and markdate < 06–12–2002
then postrule1version2
postrule2version2
postrule3
end
ruleperiod4
if markdate => 06–12–2002 and markdate < 08–12–2002
then postrule2version2
postrule3
end
```

In the next column we will continue our discussion of the rule versioning issue and discuss a

solution that is most advanced. If you are lucky your rule engine already supports this solution.

This article was originally published by BRCommunity (link).

[1] Silvie Spreeuwenberg, “Rule History and Versioning (Part 1),” *Business Rules Journal*, Vol. 8, No. 11 (Nov. 2007), URL: http://www.BRCommunity.com/a2007/b375.html