

# Enhancing the Usefulness of FOL Clauses in Rules

Wolfgang Laun

22 August 2010

## 1 Introduction

Rules in production rule systems consist of conditions (*left-hand side* or LHS) and consequences (*right-hand side* or RHS). Basically, a LHS is a conjunction of predicates selecting subsets of facts so that the rule engine may create activations of the RHS, by instantiating the RHS with all sets of the Cartesian product of the selected fact sets. As this conjunction is a truth-valued expression, it was only natural to add other Boolean expressions to the LHS. Today, such systems let you write general Boolean functions and first-order logic (FOL) formulas in addition to fact-selecting predicates, with the overall evaluation logic still being tied to the truth of a conjunction of all of these terms. This paper proposes a pragmatic extension of the evaluation strategy that may reduce the number required rules where FOL clauses are used.

## 2 The Boolean Functions in LHS Conditions

### 2.1 The FOL Functions “not”, “exists” and “forall”

These three functions express the following FOL formulas:

$$\begin{array}{l} \neg \exists x \bullet P(x) \\ \exists x \bullet P(x) \\ \forall x \bullet P(x) \end{array}$$

where the seemingly missing negated form of  $\forall$  can be written as

$$\neg \forall x \bullet P(x) \equiv \exists x \bullet \neg P(x).$$

Current implementations of rule based systems require these Boolean functions to return true for the LHS to produce rule activations. Thus, these FOL functions act as restrictive guards in the construction of the Cartesian set of fact selections. With the positive and the negated form being available, this is sufficient to express all kinds of restrictions.

## 2.2 Boolean Expression Evaluation

Arbitrary Boolean Expressions are also useful for restricting fact selection. Combined with the possibility of binding variables to attributes of (previously) selected facts, they permit writing guards of a general nature, but, consequently, their dynamic component is always based on *attribute values* of those facts.

## 3 Permissive Boolean Functions

### 3.1 Definition

Given the possibility of evaluating the Boolean functions presented in the previous section and the capability of binding any previously matched result to a LHS variable, an additional form of evaluating these functions is also possible. This *permissive evaluation* of Boolean functions is characterized by

- *not* acting as a guard, i.e., LHS evaluation continues no matter what its result, and
- the binding of a Boolean variable to receive the result.

For the case of Boolean expression evaluation it is immediately apparent that this is merely “syntactic sugar” because any Boolean expression that is not to act as a guard may simply be written where needed, without requiring to be bound to a variable. It should be noted, however, that a permissive binding to a variable could simplify subsequent patterns.

The following subsection provides the rationale for the proposed extension for the FOL functions proper.

### 3.2 Use Cases

The examples in this section use the rule notation used in Drools. Note that the combination of a Boolean function with a binding operation denotes the permissive form.

#### 3.2.1 Factoring a FOL Function into a Fact

A simple application of a permissive “exists” is shown in the following simple rule which factors a FOL

```
rule "set existence of unpaid bills"
when
    $ex : exists Bill( paid == false )
    $bf : UnpaidBills()
then
    modify( $bf ){ setValue( $ex ) }
end
```

Fact `UnpaidBills` may now conveniently be used in other rules, matching either absence or presence of unpaid bills.

Factoring of FOL functions into a simple fact is a useful abstraction, especially when the patterns in the FOL formula are more complex than in the preceding example.

### 3.2.2 Combining FOL Function Results with Fact Attributes

The following pair of rules is one of several such combinations from a program checking the consistency of a data set used in railway interlocking. While the first rule discovers the absence of a menu entry where required, the second rule fires due to the reverse error, i.e., presence of the entry where it should be absent.

```
rule "Train route start signal must have 'ZS' in menu"
when
  $e : SignalType()
  exists TrainStartGoalConn( start == $e )
  $m : SignalGraphType( element == $e,
                        menu not contains 'ZS' )
then
  Util.graphError( $e, "start of train route, no 'ZS' in menu" );
end
```

```
rule "Not train route start must not have 'ZS' in menu"
when
  $e : SignalType()
  not TrainStartGoalConn( start == $e )
  $m : SignalGraphType( element == $e,
                        menu contains 'ZS' ) from $menu
then
  Util.graphError( $e, "not start of train route, 'ZS' in menu" );
end
```

Using a permissive “exists”, both can be condensed into a single rule.

```
rule "Train route start signal must have 'ZS' in menu"
when
  $e : SignalType()
  $s : exists TrainStartGoalConn( start == $e )
  $m : SignalGraphType( element == $e, $menu : menu )
  eval( $menu.contains( "ZS" ) != $s )
then
  Util.graphError( $e, "inconsistency route start vs. 'ZS' in menu" );
end
```

### 3.2.3 Relating FOL Function Results

The next example uses the results from two FOL functions in a Boolean expression.

```
rule
when
  $e : Element( $cap : capacity )
  $p : forall ( Link( from == $e, to == $f )
               Element( this == $f, cap <= $capacity ) )
  $s : forall ( Link( from == $d, to == $e )
               Element( this == $d, cap >= $capacity ) )
then
  modify( $e ){ setSafe( $p || $s ) }
end
```

Here, the result of the “forall” functions is used to update the “safe” attribute. Notice that the update works both ways.

### 3.2.4 Permissive Form of the General Boolean Function

The general Boolean function (“eval” in Drools, “test” in Jess) can be included in the set of permissive Boolean functions. Here, however, the benefit would be small: better readability in certain (presumably rare) cases where a Boolean expression is used in more than one place of complex RHS conditions.

## 3.3 A Note on Syntax

This paper does not propose a syntax for permissive Boolean functions for any PRS. Binding variables to facts and their attributes is available, and this syntax may be used to express usage of a Boolean function in its permissive form.

Misgivings that this may lead to confusion and the erroneous usage of a binding where the guarding form of a Boolean function is intended might, however, advocate using a similar and yet distinct form for the binding operator. (In Drools, for instance, this new operator could be written as “tt :-”.)

## 4 Conclusion

The examples in the preceding section demonstrate that permissive Boolean functions are a useful addition to the expressive power of the RHS condition language. It is to be expected that their use will be rare, but this would be due to FOL functions themselves being not too frequent. But, the fact that they may help in reducing the number of rules in real-world applications should more than outweigh the relatively small implementation effort.