A Business Rule Management System based on the high-level object oriented scripting language XOTcl

Franz Wirl[1]

1 Wirtschaftsuniversität - Wien, Institute for Information Systems and New Media
franz.wirl@wu-wien.ac.at
WWW home page: http://wi.wu-wien.ac.at/nm/en/wirl

High level object oriented scripting languages like XOTcl can be perfectly used to implement Charles Forgy's Rete algorithm[1]***. An algorithm that has been developed and tested to match between more than a thousand patterns and objects. Implementing this fast algorithm into/with XOTcl will provide a fast and dynamic Rete library in XOTcl.***
An object oriented implementation allows a natural expression of rules. Object oriented interfaces to the algorithm improve its flexibility and allows usage in many different domains. [2]

Special thanks to Google, for providing this opportunity. It was a great experience to work on this project and it's a great chance for young programmers to get to learn how to do projects and to get in touch with new programming languages. Furthermore I would like to thank Univ.-Prof. Dr. Gustaf Neumann[1] for his conspirational ideas and thoughts. And last but not least I want to thank the Tcl/Tk community for giving me such a warm welcome.

Table of Contents

[1] Gustaf Neumann: http://nm.wu-wien.ac.at/nm/en/neumann

Motivation

This RETE Implementation is based on the programming language XOTcl. “XOTcl (XOTcl, pronounced exotickle) is an object-oriented scripting language based on MIT's OTcl and adds serveral novel concepts.” [3]
Using XOTcl as implementation language allows to focus on architectural design patterns of the Rete algorithm. A XOTcl native implementation can be used to use Rete within the language itself for better pattern matching of XOTcl.

Usage

Rete has been implemented as a look-a-like of the Structured Query Language (SQL). It was thought that SQL is a quite common language for developers and therefore the syntax should be known.
The implementation allows operating queries based on XOTcl classes and results are returned as XOTcl classes. Queries can return pattern matching classes, as well as to create new classes. In Listing 1 a example for a XOTcl Rete query is given. A meta class called 'View' is used to define a new class whose instances hold the query results. Class initialization is followed by a definition of involved classes as well as their variable name configuration, building pairs of classes and variable names. This definition is building the “from” part of SQL. The next “then” part defines what should happen when pattern matches. At last the “where” clause tells how patterns should be matched.

Variables defined in the “from” part can be used to match against constants and to be matched against other variables. Conditions for pattern matching are written in TCL style. Conditions are connected using the TCL and symbol “&&”. To match a variable against a condition, operators for equal “eq” and not equal “ne” are defined. If a variable is of a number type (eg integer) further math operation like less “<“, greater “>“, less than or equal “<=“ , and greater than or equal “>=“ can be performed. For further operators please, take a look at the Tcl expr manual page
http://www.tcl.tk/man/tcl8.6/TclCmd/expr.htm#M9 . If a computation of a variable should be used for pattern matching, the computation should be within square brackets. To make a computation on a computation simply enclose them again with square brackets. To get the results of a class function to extract information the class and its function have to be in square brackets. Listing 1 show how such a query looks like. Parts written in italic are areas were programmers can play around and start to code.

```
View ClassName as {
  forall {Class_object_Pairs} {
     < Attribute definition >
  } where {
     < Conditions >
  }
}
```

Listing 1 Query definition

So there are four parts where programmers can (and have to) write their query definition. In the first line the programmer has to define a name for the result class (*ClassName*). In the second line class and object pairs of the involved classes of this query have to be written (*Class_object_Pairs*). The attribute definition of the new class has to be written in line three (*Attribute definition*). And last in the fourth line the pattern matching conditions have to be defined (*Conditions*).

For example:
Get all classes of type Person whose name is shorter than three letters. First we have to know that in the "from" part in the beginning we defined an object "p" to be the variable for class "Person". To get the name of a person the class function name can be used. To get a person's name simply run: "[$p name]" to extract the name of class "Person". The result of this operation is a string and could be matched to any string operators. In a next step the length of the person's name is evaluated and matched against a constant (eg 3). Therefore we need to extract the name (as seen before) and then compute the length of this string. In a last step this computation has to be compared against a constant. The result of the Tcl function "string length *str*" returns the number of characters in *str*. So "[string length [$p name]] <= 3" could be a condition to get all persons with a name shorter or equal than three letters.

```
View WorkingPerson as {
  forall {Person p Company c} {
    my set name [$p name]
    my set company [$c name]
    my set location [$c location]
    my set salary [$p salary]
  } where {
     [$p employed] eq $c && [$c location] eq "Vienna"
  }
}
```

Listing 2 Query Example

Listing 2 defines a new class "WorkingPerson" having attributes name, company, location and salary. This class is instantiated for all objects that match the condition. The condition is true for all "Person" classes whose employed function return a class "Company" whose location function return 'Vienna'. The "then" part can also handle Tcl computations. There are four different ways how the class attributes can be set.

- set attribute to a constant value: "my set name value"
- set attribute to a variable value: "my set name $variable"
- set attribute to a computation value: "my set name [$class function]"
- set attribute as result of an expression: "my set name [expr arg1 [$class value] arg2 ...]

For example to set the attribute salary to be raised by 100 Dollar the following expression has to be build: “my set salary [expr [$p salary] + 100]”. This expression sets the attribute salary of class “WorkingPerson” to be 100 Dollar more than the original salary of this instance of class “Person”. To change the salary of an instance of class “Person” the new value has to be set to the variable *[$class set attribute value].* If you would like to raise the income of all persons represented by class “Person” who are matching the conditions the “then” part needs to have the following line *[$p set salary [expr [$p salary] + 100]]*.

To show all results the programmer has to create a loop for all instances of class “WorkingPerson” and print its attributes. Listing 3 shows how to print class “WorkingPerson”. As these results are all instances of a class “WorkingPerson” further operations and queries could be done using this object.

```
foreach w [WorkingPerson info instances] {
  puts stderr “employee '[$w set name]' company '[$w set company]'
              location '[$w set location]' with new salary '[$w set salary]' “
}
```

Listing 3 - Show results

A short class hierarchy

Most relevant classes and their functionalities are listed below. All main classes have a function print that prints information to the default output stream.

Class WorkingMemory

Class WorkingMemory is the memory for all queries. This class should be instantiated at startup, but it is automatically instantiated if it weren’t. Class WorkingMemory is responsible to save information of nodes that have already been queried. This information can be used by other nodes to operate on already calculated results. This class has two main functions:

- setInstances: saves the results of a condition
- getInstances: gets the results of a stored condition, returns an empty string if the condition hasn’t been found.
- refreshClass: refreshes the current working memory class for all conditions containing given class

Class AlphaMemory

Class AlphaMemory stores all classes and its relevant instances for a currently running condition. Each AlphaMemory class is a direct representation of the results of one condition. Main functions are:

- addClass: adds a new class to the memory
- getInstances: gets all instances of a class
- deleteInstance: remove an instance from memory

meta class View

Meta class View defines the structure of the query and creates the class for the results. The main function of this meta class is its initialization phase.

function forall

Loops over all specified classes and defines what should happen at which constraint on those classes and objects. Function “forall” makes use of the functions “runCondition” and “forallConditions”.

Class Condition and Class Conditions

The class “Conditions” is responsible to decide if a condition node is either an alpha or a beta node. A condition node is represented by class “Condition”.

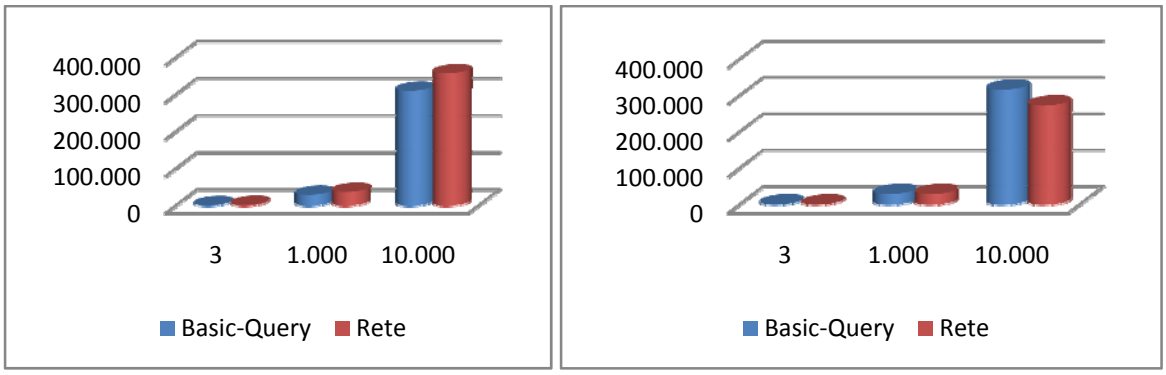
Non relevant helper Classes

All classes with no special meaning which have been used are listed. All inner classes are skipped.

- KeyListItem: a keyed list of objects and values.
- Condition_Instance_Pair: a list of “condition” and “alphaMemory” elements.

Time measurements

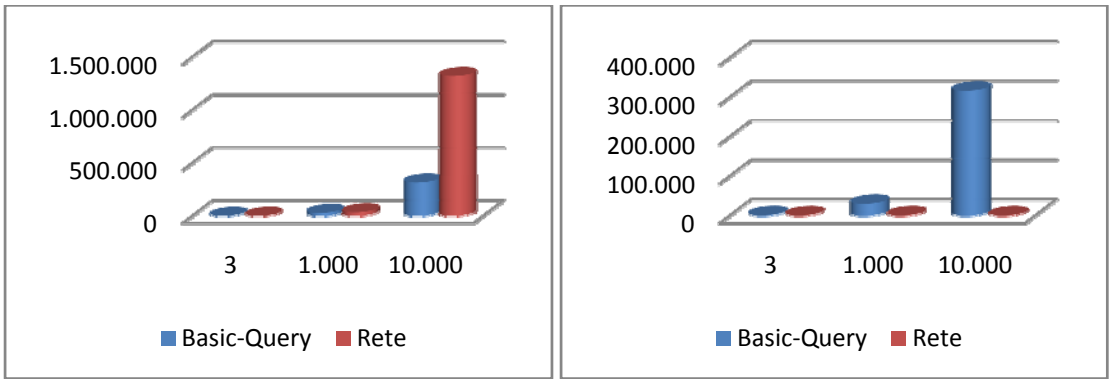
The following paragraphs are taking a close look, how well this implementation performs. As shown in the results of the measurements below, in some cases this Rete Implementation isn’t the best choice. But in many cases it makes a big difference, and helps to accelerate such kind of queries. The following figures show how many microseconds the Rete Implementation needs to run a query of three thousand or ten thousand items compared to a Basic-Query Implementation. The left part of the figures (marked with “a”) show the amount of microseconds for the first time a query is submitted. The right part “b” compares them when the query is submitted a second time. Each measurement has been done three times, and their integer median has been taken as reference.
Figure 1 show how many microseconds are needed when a query joins two classes and most items fulfill the constraints.



a b

Figure 1 Comparison of Rete vs Query Implementation when items fulfill constraints

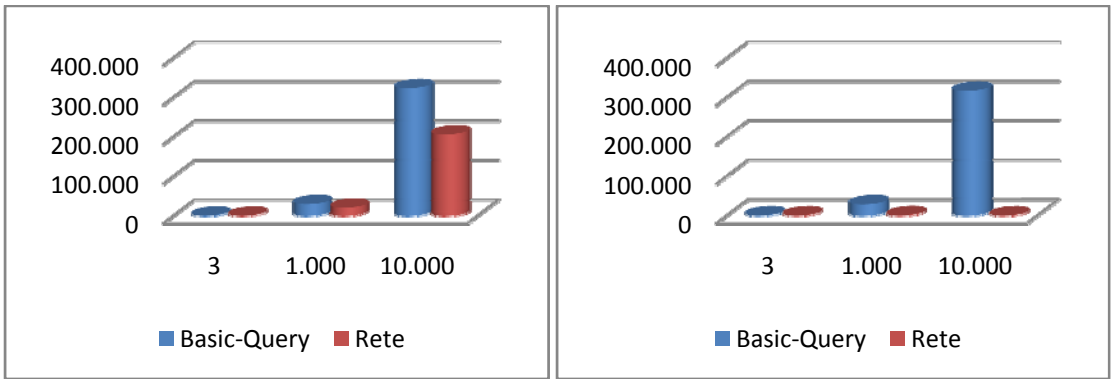
Figure 2 shows that this rete Implementation could be made more elegant when two classes are joined and most items aren't fulfilling the constraints. The first time this query executed the Rete Implementation is slower, but if the query is executed a second time the Rete Implementation is much faster.



a b

Figure 2 Comparison of Rete vs Basic-Query Implementation when items fail constraints

In last Figure 3 two queries are executed. Some constraints of the first query are reused in the second query. Left part "a" shows that the rete implementation is a bit faster if the constraints of first and second query match most items. When most items fail the constraints the rete implementation is much faster.



a b

Figure 3 Comparison of Rete vs Query Implementation, 2nd run for partly same queries

Find all numbers and queries for these measurements in Appendix.

Source Code of a full example

Find the source code of this example in Listing 4. First a model has been created. A person (class "Person") with some attributes (name, age, salary, employed) is employed by a certain company. This company (class "Company") has some attributes (name, location). After defining some facts for this model, the query can be written.

We want to know which person, older than the legal age of sixteen is working for a company in Vienna.

The result of this query should be stored in an object called "Workplace". Therefore we have to:

- Define the working memory
- Define a materialized "View" for the given class
- Show the results

A working memory is defined and initialized using the line "WorkingMemory workingMem;".
To define the class "Workplace" which stores all information about the results, the class has to be initialized by superclass "View". The start of the query is defined in line "View Workplace as { …", which starts the query definition. Define the class object pairs as described in the first example of first chapter Usage. In the condition part of the query the three conditions are defined: a.) Person old than sixteen *[$p age] >= 16* , b.) Company located in Vienna *[$c location] eq "Vienna"* and c.) The join of those classes defining that a person is working in this company *[$p employed] eq $c* .
To show the results of this query we have to loop over all representations of class "Workplace":
"foreach w [Workplace info instances] { puts … "

```
package require rete
namespace import ::rete::*

#define a global WorkingMemory that is responsible to hold all nodes
::rete::WorkingMemory workingMem;

#
# Simple Model...
#
Class Company -slots {
  Attribute name
  Attribute location
}

Class Person -slots {
  Attribute name
  Attribute age
  Attribute salary
  Attribute employed -type ::Company
}

#
# ... with a few facts ....
#

Company c1 -name KM -location Vienna
Company c2 -name AS -location US
Company c3 -name WU -location Vienna

Person p1 -name Bernd -age 31 -salary 100 -employed ::c1
Person p2 -name Gustaf -age 77 -salary 120 -employed ::c3
Person p3 -name Franz -age 22 -salary 80 -employed ::c3
Person p4 -name Jeff -age 55 -salary 120 -employed ::c2

# ok, we are done. Define a materialized View
# Workplace...
#
View Workplace as {
  forall {Person p Company c Address a} {
    my set name [$p name]
    my set company [$c name]
    my set location [$c location]
  } where {
     [$p employed] eq $c && [$c location] eq “Vienna” && [$p age] >= 16
  }
}

#
# Show the results
#
puts “print results of Query”;
foreach w [Workplace info instances] {
  puts “$w employee '[$w set name]' works for company '[$w set company]'
        located in '[$w set location]' “
}
```

Listing 4 – Full example

Bibliography

[1] C.L. Forgy, “Rete: A fast algorithm for the many pattern/many object pattern match problem,” *Artificial Intelligence*, vol. 19, Sep. 1982, pp. 17-37; http://www.sciencedirect.com/science/article/B6TYF-47X2B1P-4P/2/f31caeb8e5620fc0b89130a3dce793f3.

[2] F. Wirl, “Google Code - Summer of Code - Application Information,” *Google Code - Summer of Code - Application Information*, Apr. 2008; http://code.google.com/soc/2008/tcl/appinfo.html?csaid=2D63BF1F53E58EA1.

[3] G. Neumann, “XOTcl - Brief Description of XOTcl and ActiWeb,” *XOTcl - Brief Description of XOTcl and ActiWeb* , Aug. 2008; http://www.xotcl.org.

Appendix

	#items	Basic-Query[2]	Rete[2]
Items are relevant for join			
	3	1.587	2.366
	1.000	29.558	37.189
	10.000	310.476	357.471
rerun same Query (time for 2nd round) [Items are relevant for join]			
	3	1.185	892
	1.000	29.582	29.101
	10.000	316.894	273.377
Items are irrelevant for join			
	3	1.590	2.421
	1.000	29.173	34.979
	10.000	315.723	1.320.813
rerun same Query (time for 2nd round) [Items are irrelevant for join]			
	3	1.182	862
	1.000	30.163	1.048
	10.000	316.575	956
rerun parts of Query (time for 2nd round) [Items are relevant for join]			
	3	1.318	1.052
	1.000	30.320	21.248
	10.000	323.665	206.516
rerun parts of Query (time for 2nd round) [Items are irrelevant for join]			
	3	1.252	1.151
	1.000	29.326	1.608
	10.000	317.244	1.135

Basic body for all Queries

[2] Integer median in microsecond s after three measurements.

```
Class Company -slots {
   Attribute name
   Attribute location
 }

 Class Person -slots {
   Attribute name
   Attribute age
   Attribute salary
   Attribute employed -type ::measurement::Company
 }

 Company c1 -name KM -location Vienna
 Company c2 -name AS -location US
 Company c3 -name WU -location Vienna

 Person p1 -name Bernd -age 31 -salary 100 -employed ::measurement::c1
 Person p2 -name Gustaf -age 77 -salary 120 -employed ::measurement::c3
 Person p3 -name Franz -age 22 -salary 80 -employed ::measurement::c3
 Person p4 -name Jeff -age 55 -salary 120 -employed ::measurement::c2
```

The numbers of items are defined as listed below:

```
for { set i 1 } { $i <= 0 } { incr i } {
  [Company new -name temp -location Vienna]
 }
```

or

```
for { set i 1 } { $i <= 0 } { incr i } {
  [Company new -name temp -location SomwhereElse]
 }
```

Measurement for the first four Queries:

```
 set time_first [time {
  View Workplace as {
    forall {::measurement::Person p ::measurement::Company c} {
      my set name [$p name]
   my set company [$c name]
      my set location [$c location]
    } where {
       [$p employed] eq $c && [$c location] eq "Vienna"
    }
  }
 }]
```

Listing 5 Items are relevant / irrelevant for join

```
set time_first [time {
  View Workplace as {
    forall {::measurement::Person p ::measurement::Company c} {
      my set name [$p name]
   my set company [$c name]
      my set location [$c location]
    } where {
       [$p employed] eq $c && [$c location] eq "Vienna"
    }
  }
 }]

 set time_second [time {
  View Workplace as {
    forall {::measurement::Person p ::measurement::Company c} {
      my set name [$p name]
   my set company [$c name]
      my set location [$c location]
    } where {
       [$p employed] eq $c && [$p salary] > 90 && [$c location] eq "Vienna"
    }
  }
 }]
```

Listing 6 Parts of the first query are reused.