

Manual

Rule language of expert system shell rete4frames

Sorokin R. P.

21 september 2020



Introduction to Rete for Frames

This rete implementation uses a simple language for knowledge representation. This language is a simplification of the well known languages CLIPS (<http://clipsrules.sourceforge.net/>) and Clojure (<http://clojure.org>). Well known CLIPS examples and their implementations on this language are in Examples (<https://github.com/rururu/rete4frames/blob/master/examples>). Application program interface for embedding Rete for Frames into Java programs described in Java Interface (https://github.com/rururu/rete4frames/blob/master/doc/java_interface.md). Example of using Java interface is in Eclipse project] (<https://github.com/rururu/rete4frames/blob/master/Rete4framesEclipseTest/src/test/RetTest.java>).

The Language

The language of this rete implementation is a subset of Clojure language. Also it is similar to CLIPS language. It lacks many of CLIPS features, to mention few - multislots, slot value types, COOL. Syntax is also simplified. The rule engine of rete4frames includes only two conflict resolution strategies - depth (default) and breadth. Depth strategy means that among the rules with the highest priority fired one, which has been activated by most recent facts. Breadth strategy means that among the rules with the highest priority fired one, which has been activated by most old facts.

Application description on this language is a list consisting of four mandatory lists: templates, rules, functions and facts:

```
((templates t1 t2 ... tn )
 (rules r1 r2 ... rm )
 (functions d1 d2 ... dl )
 (facts f1 f2 ... fk ))
```

Facts can be loaded into the application from a separate file.

Frames, Facts and Patterns

The frame is a basic concept for the representation of knowledge in the language. It is used to represent the facts and patterns of facts. The frame has a type and slots. The slot has a name and a value. The frame represented as a list. First element of the list is the type of the frame. Other elements of the list are slot names and values alternately:

```
(type slot1 value1 slot2 value2 ... )
```

Values of slots can be constants and variables. Variables are represented as symbols with the question mark prefix. Constants can be of any Clojure data type. If a frame contains variables it is a pattern, otherwise it is a fact. Example of the fact:

```
(monkey location t5-7 on-top-of green-couch holding blank)
```

Example of the pattern:

```
(monkey location ?place on-top-of ?on2 holding blank)
```

Templates

The template is a description of a frame type. It is a list of the group name (type) and names of the slots. Example:

```
(monkey  
  location  
  on-top-of  
  holding)
```

Rules

The rule is a description of a transformation. It is represented by a list of the form:

```
(<name>  
  <salience>  
  <condition1>  
  <condition2>  
  ...  
  <conditionN>  
  =>  
  <function_body>)
```

The name is a symbol or a string. The salience(priority) is an integer number, positive or negative. Conditions are bare patterns, patterns with tests or negative conditions. The function body is an ordinary clojure function body which may contain, apart from everything else, the function call "asser", "retratct" and "modify". The rule "fires" when all conditions are fulfilled. In this case the function body accomplished with values for

variables obtained during pattern-matching of conditional part. First condition in the rule cannot be negative. Example of the rule:

```
(walk-holding-object 0

?goal (goal-is-to action walk-to argument1 ?place)

?monkey (monkey location ?loc on-top-of floor holding ?obj

          ((not= ?loc ?place)

           (not= ?obj blank)))

=>

(println (str "Monkey walks to " ?place " holding the " ?obj "." ))

(modify ?monkey location ?place)

(retract ?goal))
```

The part of the rule before the symbol "=>" is named "left hand side of the rule". The part of the rule after the symbol "=>" is named "right hand side of the rule".

Conditions

The condition is a bare pattern or a pattern with a test. The negative condition is a condition preceding with the symbol "not". Negative conditions must be after all positive conditions. If some condition contains a test the test is a last element of the list. The test is a predicate call or a list of tests or a vector of tests. The predicate is any Clojure function that returns true or false values, and also nil and not nil values that can be considered as true and false. The list of tests is interpreted as conjunction of tests. The vector of tests is interpreted as disjunction of tests. The condition is fulfilled if the pattern is match some fact (not match for the negative condition) and the test returns true or not nil value. If the test is absent, the pattern matching (not matching) is sufficient for the condition fulfilment. The positive (not negative) condition can be preceded with a variable which is used in a right hand side of rule in calls to functions "modify" and "retract". A value of the such variable is a fact that have been matched with the pattern. The negative condition is fulfilled if the pattern does not match any fact in the working memory and following test is true or not nil if supplied. Example of the condition:

```
(avh a color v blue h ?c5
```

```

(not= ?c5 ?c4)
(not= ?c5 ?c3)
(not= ?c5 ?c2)
(not= ?c5 ?c1)
[(= ?c5 (- ?n4 1)) (= ?c5 (+ ?n4 1))])

```

Example of the negative conditions:

```

(not thing name ladder location ?place)
(not goal-is-to action move argument1 ladder argument2 ?place)

(not edge p1 ?base-point p2 ?p4
  ((not= ?p4 ?p2) (not= ?p4 ?p3)))

```

Function Body

The function body is an ordinary clojure function body which may contain, apart from everything else, the function call "asser", "retratct" and "modify", "modify", "fact-id" and "problem-solved". It will be added with parameters list and compiled into a function object during creation of beta network. When the rule "fires" this function evaluated with values for variables obtained during pattern-matching of conditional part. The function "asser" has arguments representing a pattern of new fact that will be put into the working memory during its execution.

The function "retract" has arguments representing fact variables from the left hand side of the rule. Facts, associated with these variables, will be removed from the working memory during its execution. The function "modify" has a first argument representing a fact variable from the left hand side of the rule. Rest arguments are slots and their new values. Fact, associated with this variable, will be updated in the working memory with new values of slots during its execution. The function "fact-id" has one argument that must be a condition variable. This function returns an identifier (integer number) of a fact, which has been matched with corresponding condition during the rule activation. Then it can be used to get the fact itself using API function "frame-by-id". The function

"problem-solved" has no arguments and can be used to clear all remaining activations from the conflict set, when it become clear that, they no needed anymore.

Remark: Because Clojure functions can not have more than 20 parametrs, you can split rules with more then 20 variables in the right hand side on several rules with the same left hand side. See [zebra.clj]

(<https://github.com/rururu/rete4frames/blob/master/examples/zebra.clj>) example.

Functions

The functions section contains ordinary clojure function definitions optionally prepended with namespace definition (ns) See examples of function definitions in example [waltz.clj] (<https://github.com/rururu/rete4frames/blob/master/examples/waltz.clj>).

Application Program Interface (API)

1.Function app

```
(app <mode> <application-file>)  
(app <mode> <application-file> <facts-file>)
```

Run the application.

- application-file (string) - a path to a file containing a list of templates, rules, functions and facts.
- facts-file (string) - a path to a file containing a list of facts. If used, this list of facts replaces the list of facts from the .
- mode (string):
 - "run" - run the application synchronously, that is assert first fact and fire rules then assert second fact and fire rules and so on.
 - "trace" - same with a tracing of creation of the rete network and tracing of firing rules and assertion and retraction of facts.

- "trace-long" - same verbose.
- "step" - same as "trace" with a stop after firing one rule. Further execution can be continued by calling functions (fire N) or (fire) where N - a number of steps (firings of one rule). If N is omitted then fire till the very end.

In the modes with the tracing in a root directory are created three files:

- alpha-net-plan.txt - describes alpha part of the rete network.
- beta-net-plan.txt - describes beta part of the rete network.
- alpha-beta-links.txt - describes the links between alpha part and beta part of the rete network.

2.Function fact-list

```
(fact-list)
(fact-list '<type>)
```

Returns a list of facts (of type if supplied) in a working memory.

3.Function facts

```
(facts)
(facts '<type>)
```

Prints a list of facts (of type if supplied) in a working memory.

4.Function ppr

```
(ppr <type>)
```

Pretty prints facts of a specific type or all facts (= :all) in a working memory.

5. Function run-with

```
(run-with <mode> <templates> <rules> <facts>)
```

Create the rete network and run using given mode.

- templates - a list of templates.

- rules - a list of rules.
- facts - a list of facts.

6. Functions trace/untrace

```
(trace)
(untrace)
```

Switches on/off tracing.

7. Functions step

```
(step)
(step N)
```

Fires rules 1 time (N times if supplied).

8. Functions run-synch/run-async

```
(run-with-mode <mode> <truff>)
(run-with-mode <mode> <trufs> <facts>)
```

Create the rete network and run run using given mode. Arguments are: - "run" or "trace" or "step", - a list of a form: ((templates ..) (rules ..) (functions ..) (facts ...)), - a list of a form: ((templates ..) (rules ..) (functions ..)), - a list of facts.

9. Function create-rete

```
(create-rete <templates> <rules>)
```

Create the rete network and clear the working memory. Arguments are the same as for run-with function.

10. Function assert-frame

```
(assert-frame <fact>)
```

Assert a fact into the working memory.

11. Function retract-fact

```
(retract-fact <fact-id>)
```

Retract a fact from the working memory.

- fact-id (integer) - index of the fact in the working memory.

12. Function modify-fact

```
(modify-fact <fact-id> <slot-value-map>)
```

Modify a fact in the working memory.

- fact-id (integer) - index of the fact in the working memory.
- slot-value-map - map of slots and their values. During modification the old fact is deleted and a new fact created with the changed values of slots according to map.

13. Function fire

```
(fire <number>)  
(fire)
```

Fire a number of active rules. If the number is omitted fire till the very end.

14. Function reset

```
(reset)
```

Clear and initialize the working memory.

15. Function strategy-depth

```
(strategy-depth)
```

Set conflict resolution strategy to depth.

16. Function strategy-breadth

```
(strategy-breadth)
```

Set conflict resolution strategy to breadth.

17. Function run-loaded-facts

```
(run-loaded-facts <path>)
```

Load facts from path, assert all of them into working memory and run,

- path - string representing a path to a file.

18. Function only-load-facts

```
(only-load-facts <path>)
```

Load facts from path and assert all of them into working memory,

- path - string representing a path to a file.

19. Function save-facts

```
(save-facts <path>)
```

```
(save-facts <types> <path>)
```

Save all facts or facts of types to a file on a path in a format suitable to load,

- path - string representing a path to a file,
- types - collection of types of facts.

20. Function slot-value

```
(slot-value <slot> <fact>)
```

Returns value of a slot of a fact,

- slot - symbol representing a slot of a fact,

- fact - list representing a fact (as of item of result of the function fact-list).

21. Function facts-with-slot-value

```
(facts-with-slot-value <slot> <function> <value>)
```

```
(facts-with-slot-value <type> <slot> <function> <value>)
```

```
(facts-with-slot-value <type> <slot> <function> <value> <facts>)
```

Returns list of all facts or facts of type with slot values for which (function slot-value value) = true/not nil,

- slot - symbol representing a slot of a fact,
- function - symbol - predicate for selection of facts,
- type - symbol representing a type of a fact,
- value - any value,
- facts - list - preselected facts.

Examples:

```
(facts-with-slot-value 'PersonalData 'name = "Alice")
```

```
(facts-with-slot-value 'BloodPressure 'systolic > 140)
```

```
(facts-with-slot-value 'BloodPressure 'systolic > 140 (facts-with-slot-value  
'BloodPressure 'diastolic > 90))
```

22. Function frame-by-id

```
(frame-by-id <fact-id>)
```

Returns a fact,

- fact-id - identifier (integer-number) of a fact as frame.

While embedding Rete for Frames into your code you can use other functions. See [source code] (<https://github.com/rururu/rete4frames/blob/master/src/rete/core.clj>).

Example: auto repairing

((templates

(working-state

engine)

(spark-state

engine)

(charge-state

battery)

(rotation-state

engine)

(symptom

engine)

(repair

advice)

(stage

value))



(rules

(normal-engine-state-conclusions

8

(working-state engine normal)

=>

(asser repair advice "No repair needed.")

(asser spark-state engine normal)

(asser charge-state battery charged)

(asser rotation-state engine rotates))

(unsatisfactory-engine-state-conclusions

8

(working-state engine unsatisfactory)

=>

(asser charge-state battery charged)

(asser rotation-state engine rotates))

(determine-engine-state

8


(stage value diagnose)

(not working-state engine ?ws)

(not repair advice ?a)

=>

(if (a/yes-or-no "Does the engine start?")



(if (a/yes-or-no "Does the engine run normally?")

(asser working-state engine normal)

(asser working-state engine unsatisfactory))

(asser working-state engine does-not-start)))

(determine-rotation-state

0

(working-state engine does-not-start)

(not rotation-state engine ?rs)

(not repair advice ?a)

=>

(if (a/yes-or-no "Does the engine rotate?")

(do

(asser rotation-state engine rotates)

(asser spark-state engine irregular-spark))

(do

(asser rotation-state engine does-not-rotate)

(asser spark-state engine does-not-spark))))))


(determine-sluggishness

3

(working-state engine unsatisfactory)

(not repair advice ?a)

=>



```
(if (a/yes-or-no "Is the engine sluggish?")  
  (asser repair advice "Clean the fuel line."))))
```

```
(determine-misfiring
```

```
2
```

```
(working-state engine unsatisfactory)
```

```
(not repair advice ?a)
```

```
=>
```

```
(if (a/yes-or-no "Does the engine misfire?")
```

```
(do
```

```
  (asser repair advice "Point gap adjustment.")
```

```
  (asser spark-state engine irregular-spark))))
```

```
(determine-knocking
```

```
1
```

```
(working-state engine unsatisfactory)
```

```
(not repair advice ?a)
```

```
=>
```


```
(if (a/yes-or-no "Does the engine knock?")
```

```
  (asser repair advice "Timing adjustment."))))
```

```
(determine-low-output
```

```
0
```

```
(working-state engine unsatisfactory)
```

(not symptom engine ?se)

(not repair advice ?a)

=>

(if (a/yes-or-no "Is the output of the engine low?")

(asser symptom engine low-output)

(asser symptom engine not-low-output)))

(determine-gas-level

0

(working-state engine does-not-start)

(rotation-state engine rotates)

(not repair advice ?a)

=>

(if (not (a/yes-or-no "Does the tank have any gas in it?"))

(asser repair advice "Add gas.)))

(determine-battery-state

0

(rotation-state engine does-not-rotate)

(not charge-state battery ?cs)

(not repair advice ?a)

=>

(if (a/yes-or-no "Is the battery charged?")

(asser charge-state battery charged)



(do

(asser repair advice "Charge the battery.")

(asser charge-state battery dead))))

(determine-point-surface-state-1

8

(working-state engine does-not-start)

(spark-state engine irregular-spark)

(not repair advice ?a)

=>

(condp = (a/ask "What is the surface state of the points?" '[normal burned contaminated])

'burned (asser repair advice "Replace the points.")

'contaminated (asser repair advice "Clean the points.")

'normal))

(determine-point-surface-state-2

0

(symptom engine low-output)

(not repair advice ?a)

=>

(condp = (a/ask "What is the surface state of the points?" '[normal burned contaminated])

'burned (asser repair advice "Replace the points.")

'contaminated (asser repair advice "Clean the points.")

'normal))



```
(determine-conductivity-test
```

```
0
```

```
(working-state engine does-not-start)
```

```
(spark-state engine does-not-spark)
```

```
(charge-state battery charged)
```

```
(not repair advice ?a)
```

```
=>
```

```
(if (a/yes-or-no "Is the conductivity test for the ignition coil positive?")
```

```
  (asser repair advice "Repair the distributor lead wire.")
```

```
  (asser repair advice "Replace the ignition coil.))))
```

```
(no-repairs
```

```
-8
```

```
(stage value diagnose)
```

```
(not repair advice ?a)
```

```
=>
```

```
(asser repair advice "Take your car to a mechanic.))
```

```
(system-banner
```

```
0
```

```
?s (stage value start)
```

```
=>
```

```
(println "The Engine Diagnosis Expert System")
```



```
(println)
```

```
(modify ?s value diagnose))
```

```
(print-repair
```

```
-8
```

```
(repair advice ?a)
```

```
=>
```

```
(println)
```

```
(println (str "Suggested Repair: " ?a))
```

```
(println)))
```

```
(functions
```

```
(ns a)
```

```
(defn ask [q ops]
```

```
  (println (str q " " ops))
```

```
  (if-let [a (read)]
```

```
    (if (some #{a} ops)
```

```
      a
```

```
      (ask q ops))))
```

```
(defn yes-or-no [q]
```

```
  (if (= (ask q '[yes no]) 'yes)
```

```
    true
```



```
false)))
```

```
(facts
```

```
(stage value start)))
```

More examples see (<https://github.com/rururu/rete4frames/tree/master/examples>)