DEVICE Incremental Matching Algorithms

This page includes the details concerning the structure (objects), the behavior (matching algorithms) and the relationships (links) between the complex event network nodes.

```
class event
class variables
   instance_of      : event_meta_class
   is_a             : object
attributes
   signal           : {<two_input_event>|<active_rule>}
   anti_signal      : {<two_input_event>|<production_rule>}
   out_token        : [{<plog>}]
class one_input_event
class variables
   instance_of      : event_meta_class
   is_a             : event
class primitive_event
class variables
   instance_of      : event_meta_class
   is_a             : one_input_event
attributes
   method           : <method>
   class            : <class>
   when             : before|after
   signal           : {<logical_event>|<two_input_event>|<active_rule>}
   anti_signal      : {<logical_event>|<two_input_event>|<production_rule>}
   out_token        : [<msg_recipient>,<plog>]
methods
signal([OID,PARAM])
        OUT_TOKEN := [OID,PARAM]
        % Note that one of signal(self) or anti_signal(self) (but not both) is null
        ∀ EID ∈ signal(self)
            signal([self,OUT_TOKEN]) ⇒ EID
        end ∀
        ∀ EID ∈ anti_signal(self)
            anti_signal([self,OUT_TOKEN]) ⇒ EID
        end ∀
return()
class logical_event
class variables
   instance_of      : event_meta_class
   is_a             : one_input_event
attributes
   signal_by        : {<primitive_event>}
   anti_signal_by   : {<primitive_event>}
   condition        : <plog>
   in_token         : [<msg_recipient>,<plog>]
   out_token        : [<msg_recipient>]|[<msg_recipient>,<plog>]
methods
signal([P_EID,IN_TOKEN])
   if match([IN_TOKEN],condition(self)) then
     if length(out_token(self))=1 then
       OUT_TOKEN := [OID]
     else
       OUT_TOKEN := [OID,PARAM]
     end if
     ∀ EID ∈ signal(self)
       signal([self,OUT_TOKEN]) ⇒ EID
     end ∀
```

```
  end if
return()
anti_signal([P_EID,IN_TOKEN])
  if match([IN_TOKEN],condition(self)) then
    if length(out_token(self))=1 then
      OUT_TOKEN := [OID]
    else
      OUT_TOKEN := [OID,PARAM]
    end if
    ∀ EID ∈ anti_signal(self)
      anti_signal([self,OUT_TOKEN]) ⇒ EID
    end ∀
  end if
return()
```

class two_input_event

```
class variables
  instance_of          : event_meta_class
  is_a                 : event
attributes
  left_signal_by       : {<event>}
  right_signal_by      : {<event>}
  left_anti_signal_by  : {<event>}
  right_anti_signal_by : {<event>}
  lr_join              : {<plog>}
  rl_join              : {<plog>}
  left_in_token        : [{<plog>}]
  right_in_token       : [{<plog>}]
  left_in_memory       : {<left_in_token>}
  right_in_memory      : {<right_in_token>}
  left_virtual         : virtual({true|false},<plog>)
  right_virtual        : virtual({true|false},<plog>)
  left_counter         : on|off
  right_counter        : on|off
methods
check_lr([TOKEN_LIST])
  CHECK := true
  ∀ COND ∈ lr_join(self)
    if ¬match([TOKEN_LIST],COND) then
      CHECK := false
      exit ∀
    end if
  end ∀
return(CHECK)
check_rl([TOKEN_LIST]) % The same algorithm, with "rl_join" attribute in place of "lr_join".
```

class positive_event

```
class variables
  instance_of        : event_meta_class
  is_a               : two_input_event
methods
signal([C_EID,IN_TOKEN])
  if C_EID ∈ left_signal_by(self) then
    if token(IN_TOKEN,COUNT) ∈ left_memory(self) then
      left_memory(self) := (left_memory(self) - {token(IN_TOKEN,COUNT)}) ∪
                    {token(IN_TOKEN,COUNT+1)}
    else
      left_memory(self) := left_memory(self) ∪ {token(IN_TOKEN,1)}
      ∀ token(TOKEN,_) ∈ right_memory(self)
        if check_lr([[IN_TOKEN,TOKEN]] ⇒ self then
          OUT_TOKEN := match([IN_TOKEN,TOKEN],out_token(self))
          ∀ EID ∈ signal(self)
            signal([self,OUT_TOKEN]) ⇒ EID
          end ∀
        end if
      end ∀
```

```
    end if
  else    % The same algorithm, with the "right" keyword in the place of "left" and vice-versa
  end if
return()
anti_signal([C_EID,IN_TOKEN])
  if C_EID ∈ left_anti_signal_by(self) then
    ∀ token(IN_TOKEN,COUNT) ∈ left_memory(self) ∧ COUNT>1
      left_memory(self) := (left_memory(self) - {token(IN_TOKEN,COUNT)}) ∪
                    {token(IN_TOKEN,COUNT-1)}
    end ∀
    ∀ token(IN_TOKEN,1) ∈ left_memory(self)
      left_memory(self) := left_memory(self) − {token(IN_TOKEN,1)}
      OUT_TOKEN := match([IN_TOKEN],out_token(self))
      ∀ EID ∈ anti_signal(self)
        anti_signal([self,OUT_TOKEN]) ⇒ EID
      end ∀
    end ∀
  else    % The same algorithm, with the "right" keyword in the place of "left" and vice-versa
  end if
return()
class negative_event
class variables
  instance_of        : event_meta_class
  is_a               : two_input_event
methods
signal([C_EID,IN_TOKEN])
  if C_EID ∈ left_signal_by(self) then
    if token(IN_TOKEN,COUNT) ∈ left_memory(self) then
      left_memory(self) := (left_memory(self) - {token(IN_TOKEN,COUNT)}) ∪
                    {token(IN_TOKEN,COUNT+1)}
    else
      left_memory(self) := left_memory(self) ∪ {token(IN_TOKEN,1)}
      MATCH := false
      ∀ token(TOKEN,_) ∈ right_memory(self)
        if check_lr([[IN_TOKEN,TOKEN]]) ⇒ self then
          MATCH := true
          exit ∀
        end if
      end ∀
      if ¬MATCH then
        OUT_TOKEN := match([IN_TOKEN],out_token(self))
        ∀ EID ∈ signal(self)
          signal([self,OUT_TOKEN]) ⇒ EID
        end ∀
      end if
    end if
  else
    if token(IN_TOKEN,COUNT) ∈ right_memory(self) then
      right_memory(self) := (right_memory(self) - {token(IN_TOKEN,COUNT)}) ∪
                    {token(IN_TOKEN,COUNT+1)}
    else
      right_memory(self) := right_memory(self) ∪ {token(IN_TOKEN,1)}
      ∀ token(TOKEN,_) ∈ left_memory(self)
        if check_rl([[IN_TOKEN,TOKEN]]) ⇒ self then
          OUT_TOKEN := match([TOKEN],out_token(self))
          ∀ EID ∈ anti_signal(self)
            anti_signal([self,OUT_TOKEN]) ⇒ EID
          end ∀
        end if
      end ∀
    end if
```

```
  end if
return()
anti_signal([C_EID,IN_TOKEN])
  if C_EID ∈ left_anti_signal_by(self) then
    ∀ token(IN_TOKEN,COUNT) ∈ left_memory(self) ∧ COUNT>1
      left_memory(self) := (left_memory(self) - {token(IN_TOKEN,COUNT)}) ∪
                    {token(IN_TOKEN,COUNT-1)}
    end ∀
    ∀ token(IN_TOKEN,1) ∈ left_memory(self)
      left_memory(self) := left_memory(self) - {token(IN_TOKEN,1)}
      OUT_TOKEN := match([IN_TOKEN],out_token(self))
      ∀ EID ∈ anti_signal(self)
        anti_signal([self,OUT_TOKEN]) ⇒ EID
      end ∀
    end ∀
  else
    ∀ token(IN_TOKEN,COUNT) ∈ right_memory(self) ∧ COUNT>1
      right_memory(self) := (right_memory(self) - {token(IN_TOKEN,COUNT)}) ∪
                    {token(IN_TOKEN,COUNT-1)}
    end ∀
    ∀ token(IN_TOKEN,1) ∈ right_memory(self)
      right_memory(self) := right_memory(self) - {token(IN_TOKEN,1)}
      ∀ token(TOKEN,_) ∈ left_memory(self)
        if check_rl([[IN_TOKEN,TOKEN]]) ⇒ self then
          MATCH := false
          ∀ token(R_TOKEN,_) ∈ right_memory(self)
            if check_lr([[R_TOKEN,TOKEN]]) ⇒ self then
              MATCH := true
              exit ∀
            end if
          end ∀
          if ¬MATCH then
            OUT_TOKEN := match([TOKEN],out_token(self))
            ∀ EID ∈ signal(self)
              signal([self,OUT_TOKEN]) ⇒ EID
            end ∀
          end if
        end if
      end ∀
    end ∀
  end if
return()
class active_rule
class variables
  instance_of        : active_rule_manager
  is_a               : object
  ordered_rule_list  : {[<active_rule>,<integer>]}
attributes
  signal_by          : {<event>}
  in_token           : [{<plog>}]
methods
signal([EID,IN_TOKEN])
  if match([IN_TOKEN],condition(self)) then
    fire([self,match([IN_TOKEN],action(self))]) ⇒ self
  end if
return()
fire([RuleOID,ACTION])
  ∀ GOAL ∈ ACTION
    GOAL
  end ∀   % Immediate Event-Condition and Condition-Action Coupling modes.
return()
```

```
class production_rule
class variables
   instance_of      : production_rule_manager
   is_a             : active_rule
   refractoriness   : on|off
   recency          : on|off
   specificity      : on|off
   conflict_set     : {[<production_rule>,<plog>,<plog>]}
   task             : <atom>
attributes
   anti_signal_by   : <two_input_event>
   no_of_events     : <integer>
methods
signal([EID,IN_TOKEN])
   schedule([self,IN_TOKEN,match([IN_TOKEN],action(self))]) ⇒ production_rule
return()
anti_signal([EID,IN_TOKEN])
   de_schedule([self,IN_TOKEN]) ⇒ production_rule
return()
class active_rule_manager
class variables
   instance_of      : meta_rule_manager
   is_a             : class
class production_rule_manager
class variables
   instance_of      : meta_rule_manager
   is_a             : active_rule_manager
methods
new(Rule)
   RuleOID := compile_rule(Rule)
   if specificity(self)=on then
     ordered_rule_list(self) := sort(ordered_rule_list(self) ∪
                      {[RuleOID,no_of_events(RuleOID)]})
   else
     ordered_rule_list(self) := ordered_rule_list(self) ∪
                      {[RuleOID,no_of_events(RuleOID)]}
   endif
return(RuleOID)
schedule([RID,TOKEN,ACTION])
   if recency(self)=on then
     conflict_set(self) := {[RID,TOKEN,ACTION]} ⊕ conflict_set(self)
   else
     conflict_set(self) := conflict_set(self) ⊕ {[RID,TOKEN,ACTION]}
   endif
return()
de_schedule([RID,TOKEN])
   conflict_set(self) := conflict_set(self) - {[RID,TOKEN,_]}
return()
end_of_transaction
   select_and_fire ⇒ production_rule  % Rules are selected and executed at the end of the transaction
...          % The rest of the operations that need to be done by the database system at the end of the transaction
return()
select_and_fire
   TOKEN := top(conflict_set(self) ∩ ordered_rule_list(self))
   if refractoriness(self)=on then
     conflict_set(self) := conflict_set(self) - TOKEN
   end if
   TOKEN ≈ [RID,_,ACTION]      % The ≈ operator analyses a token into its constituent values
   fire([RID,ACTION]) ⇒ RID
   if conflict_set(self)≠∅
     select_and_fire ⇒ production_rule
   endif
return()
```

In the previous class definitions the following symbols are used that cannot be further analysed:

`<plog>`	`::=`	*any valid Prolog term*
`<atom>`	`::=`	*any valid Prolog atom (string)*
`<msg_recipient>`	`::=`	*the OID of the message recipient object*
`<class>`	`::=`	*a user-defined class of the database schema*
`<method>`	`::=`	*a recognisable method of a class of the database schema*

Furthermore, the following classes are supposed to exist in the system:

`object`	:	*the top class of the class hierarchy*
`event_meta_class, rule_meta_class`	:	*meta-classes to support functionality for event and rule objects*

Finally, the following functions are also used without detailed description:

`match(ListOfVariables,PrologTerm)`: replaces the occurrences of variables inside `PrologTerm` with values of variables in `ListOfVariables`. In Prolog this is a straightforward operation.

`top(List)`: returns the first element of `List`.

`compile_rule(Rule)`: compiles the rule `Rule` according to section 4 and returns its OID. More detailed algorithms for this function in [3].

`sort(ListOfRules)`: sorts in descending order the list of tuples `ListOfRules` according to the value of the second tuple position.