www.francetelecom.com/

(Unrestricted)

Research & Development

A quick tutorial on the development of the JADE Semantics Add-on demonstration

Authors: Vincent PAUTRET, Vincent LOUIS

Version: 1.1

France Telecom
Research & Development

2, av. Pierre Marzin
22307 Lannion Cedex — France
Phone: +33 296 051 111

In this document, we focus on the specific aspects to develop semantic agents upon the framework provided by the JADE Semantics Add-on. We do not present the GUI aspects or classical aspects of programming jade agents. For this purpose, you can refer to "*Jade Programming for Beginners*" or "*Jade Programmer's Guide*".

To program the semantic agents of this demonstration, we mainly developed observers, filters for the assertion and querying operations on the Knowledge base and ontological actions related to the specific domain of the temperature variations. The development of a semantic agent does generally not involve explicit management of the messages exchanged between the agents. Rather, these tasks are dealt with by the generic algorithms provided in the `SemanticAgentCapabilities` class.

The code is primarily gathered in the agent's `capabilities` class.

1 Temperature definition

The `FiltersDefinition` class is very useful to gather filters on the knowledge base that deal with the management (i.e. assertion and querying) of specific predicates. In this demonstration, we have developed a `SingleNumValueDefinition` class (extending the `FiltersDefinition` class) that deals with assertion and querying facts on single numeric value predicates. Here, we use this class to define the `temperature`, and `temperature_gt` predicates. This class is shared by all the semantic agents of the demonstration: the sensor agents (`SensorAgent` class), the display agent (`DisplayAgent` class) and the son agent (`ManAgent` class).

Each filter is basically defined using the `defineFilter` method. For each `assertFilter`, the `KBAssertFilterAdapter` class is used, and for each `queryFilter`, the `KBQueryFilterAdpater` is used. According to needs, the methods `doApply` and `afterAssert` can be overridden for the assert filters and the `apply` method can be overridden for the query filters.

Temperature filters handle two different predicates:

- `(temperature X)`, where `X` is a numeric value, meaning the current temperature value is `X`,
- `(temperature_gt X)`, where `X` is a numeric value, meaning the current temperature value is (strictly) greater than `X`.

The assert filters ensure that at most one temperature value can be asserted into the knowledge base with the `temperature` predicate, and maintain a consistence of all asserted `temperature_gt` and `temperature` facts.
The query filters compute the truth value of queried `temperature_gt` facts with respect to the `temperature_gt` and `temperature` facts belonging to the knowledge base.

<u>Example of an assert filter:</u>

// Creates a new instance of `KBAssertFilterAdapter` with the appropriate pattern
```
defineFilter(new KBAssertFilterAdapter("(B ??agent"+VALUE_X_PATTERN + ")"){
```
// Overrides the `doApply` method:
// if the formula to be asserted is a temperature value, then the current agent
// knowledge on temperature is deleted and the formula is returned, so that
// this new value will be finally asserted by the raw assertion mechanism
// (unless it is trapped by another filter).

```
        public Formula doApply(Formula formula) {
            if ((myKBase.query(formula) != null)) {
                return new TrueNode();
            }
            cleanKBase((FilterKBaseImpl) myKBase);
            return formula;
        }
});
```

Example of a query filter :

```
// Creates a new instance of KBQueryFilterAdapter with the appropriate pattern:
defineFilter(new KBQueryFilterAdapter("(B ??agent " +VALUE_GT_X_PATTERN+")") {
// then apply method is overridden because some tests are done during the matching
public QueryResult apply(Formula formula, Term agent) {
  // By default the result is false and the list of solutions is set to null
  QueryResult queryResult = new QueryResult();
  try {
    // test the agent
    SLPatternManip.set(pattern, "agent", agent);
    MatchResult applyResult = SLPatternManip.match(pattern, formula);
    // the agent should be the current one and the metavariable X (see the pattern) should be a
    // constant
    if (applyResult != null && applyResult.getTerm("X") instanceof Constant) {
          // Gets the queried temperature from the incoming formula
      Long queriedValue = ((Constant)applyResult.getTerm("X")).intValue();
          // Queries the knowledge base about the current temperature
      ListOfTerm queryRefResult = myKBase.queryRef((IdentifyingExpression)
                  SLPatternManip.instantiate(IOTA_VALUE, "agent", applyResult.getTerm("agent")));
         // If a temperature value has been found in the Belief base
      if (queryRefResult != null && queryRefResult.size() != 0 ) {
       if ( ((Constant)queryRefResult.get(0)).intValue().longValue() > queriedValue.longValue() )
{
             // If the found temperature is greater than the queried temperature,
             // then the filter returns an empty ListOfMatchResult.
             queryResult.setResult(new ListOfMatchResults());
       }
      }
      else {
             // Queries the knowledge base about a relative temperature
        queryRefResult = myKBase.queryRef((IdentifyingExpression)
            SLPatternManip.instantiate(IOTA_VALUE_GT, "agent",  applyResult.getTerm( "agent")));
         if (queryRefResult != null && queryRefResult.size() != 0 ) {
                         if     (((Constant)queryRefResult.get(0)).intValue().longValue()     >=
queriedValue.longValue() ) {
             // If a relative temperature has been found and if it is greater than the queried
             // temperature, then returns an empty ListOfMatchResult as above
               queryResult.setResult(new ListOfMatchResults());
                         }
         }
      }
     // The boolean value of the result is set to true meaning the filter is applicable
      queryResult.setFilterApplied(true);
    }
  } catch (Exception e) {
        e.printStackTrace();
```

```
 }
// Returns the result
 return queryResult;
}
```

// The value of the VALUE_GT_X_PATTERN formula depends of the of the VALUE_X_PATTERN, the VALUE_GT_X_PATTERN and the NOT_ VALUE_GT_X_PATTERN. So, these three formula are added in the set of formulae that trigger the observer on VALUE_GT_X_PATTERN.

```
 public void getObserverTriggerPatterns(Formula formula, Set set) {
  try {
    MatchResult applyResult = SLPatternManip.match(pattern, formula);
    if (applyResult != null && applyResult.getTerm("X") instanceof Constant) {
            set.add(VALUE_X_PATTERN);
            set.add(VALUE_GT_X_PATTERN);
            set.add(NOT_VALUE_GT_X_PATTERN);
     }
  }catch (SLPatternManip.WrongTypeException wte) {
    wte.printStackTrace();
  }
}
```

2 Sensor agent

The sensor agent is a very simple semantic agent that loads the previous temperature definition in the `setupKbase` method of its `SemanticCapabilities` class (which is the `SensorCapabilities` class).

2.1 Defining the sensor behaviour

The behaviour of the sensor agent is customized by overriding several methods of the `StandardCustomizationAdapter` class.

```
public void setupStandardCustomization() {
        setMyStandardCustomization( new StandardCustomizationAdapter() {
            // This method prevents the agent from adopting any belief about a predicate of the
            // temperature domain from any agent
   public boolean acceptBeliefTransfer(Formula formula, Term agent) {
      return (SLPatternManip.match(temperatureDefinition.VALUE_X_PATTERN, formula)== null)
      && (SLPatternManip.match(temperatureDefinition.NOT_VALUE_X_PATTERN, formula)== null)
      && (SLPatternManip.match(temperatureDefinition.VALUE_GT_X_PATTERN, formula)== null)
    && (SLPatternManip.match(temperatureDefinition.NOT_VALUE_GT_X_PATTERN, formula) == null); }
}));

            // This method is called by the semantics framework to handle calls for proposal
            // received by the sensor. Here, this method returns the value of the sensor precision
            // or null if the sensor does not have any precision.
   public ListOfTerm handleCFPAny(Variable variable, Formula formula, ActionExpression action, Term agent) {
           if (SLPatternManip.match(SLPatternManip.fromFormula("(precision ??X)"), formula) !=
           null ) {
              return myKBase.queryRef(new AnyNode(variable, formula));
           }
           else {
             return null;
           }
   }

           // This method is called by the semantics framework when the agent receives a subscribe
           // message. Here this notification is only used to change the colour of the sensor.
   public void notifySubscribe(Term subscriber, Formula obsverved, Formula goal) {
           ((SensorAgent)myAgent).setSubscribed(true);
   }

           // This method is called by the semantics framework when the agent receives an
           // unsubscribe message (which is in fact an Inform message with the proper content)
   public void notifyUnsubscribe(Term subscriber, Formula obsverved, Formula goal) {
           ((SensorAgent)myAgent).setSubscribed(false);
   }
});
```

2.2 Registering to the DF Semantic Agent

When clicking on the button of a sensor agent, it registers or deregisters to the DF Semantic Agent of the demonstration (which is a semantic agent created for the purpose of the demonstration, and which plays the role of a semantic version of a standard DF agent). This simply consists in sending an Inform message stating the sensor is active or inactive (see the `actionPerformed` method of the GUI button in the `SensorAgent` class). This DF agent then stores in its knowledge base the current state of each sensor. Note that the DF semantic agent is a generic `SemanticAgentBase` without any specific code.

2.3 Slider variations

To take into account the variations of the slider, the Sensor Agent GUI (`SensorAgent` class) updates the Knowledge base of the agent by generating an internal semantic event, each time the slider changes its location:

```
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent evt) {
            getSemanticCapabilities().getSemanticInterpreterBehaviour().interpret("(B ??agent (=
(iota ?x (temperature ?x)) " + (double)slider.getValue()/tenPowerPrecision + ")) ");
    }
});
```

This new belief assertion will be catched by the filters of the `SingleNumValueDefinition` class, so that the new temperature value will be properly processed.

3 Display agent

The display agent is a very simple semantic agent that loads the previous temperature definition in the `setupKbase` method of its `SemanticCapabilities` class (which is the `DisplayCapabilities` class).

3.1 Subscription

When an item (Least/Most) is chosen in the combo list, a `Query-Ref` is sent to the DF Agent:

```
getSemanticCapabilities().sendCommunicativeAction(getSemanticCapabilities().createQueryRef(SUBSC
RIBE_DF_IRE, dfagent));
```

As a result (when receiving the answer from the DF agent), the Display knows the list of the registered sensors. Then, it sends a `Call-For-Proposal` to each sensor:

```
getSemanticCapabilities().sendCommunicativeAction(getSemanticCapabilities().createCFP(
(ActionExpression(CFP_ACTION,
            "agent", getSemanticCapabilities().getAgentName(),
            "receiver", sensors.elmement(i)),
            CFP_IRE,
            sensors.element(i)));
```

To select the sensor, the method `handleProposal` (see the `DisplayCapabilities` class) is overridden. The agent handles the proposal only if the action is an `InformRef` on temperature and if the condition relates to a precision. According to the requested precision, the appropriate sensor is then selected.

Finally, a `Subscribe` message is sent to the selected sensor agent, so that it will inform the display of each future temperature changes:

```
getSemanticCapabilities().sendCommunicativeAction(getSemanticCapabilities().createSubscribe(SUBS
CRIBE_IRE, selectedAgent));
```

3.2 Belief Transfer

Like for the sensor agent, the `acceptBeliefTransfer` method is overridden in order to specify a particular belief transfer behaviour for the display agent.

```
setMyStandardCustomization(new StandardCustomizationAdapter() {
// This method prevents the agent from adopting any belief about a predicate of the
// temperature domain from any agent but the subscribed sensor agent
   public boolean acceptBeliefTransfer(Formula formula, Term agent) {
      Formula toBeTested = formula;
       // Is the incoming formula an Equals Formula with an Identifying expression and a formula
       MatchResult match = SLPatternManip.match(SLPatternManip.fromFormula("(= ??ire ??value)"),
formula);
      if (match != null) {
         try {
            // Take the identifying expression as the formula to be tested
            toBeTested = ((IdentifyingExpression)match.getTerm("ire")).as_formula();
         } catch (SLPatternManip.WrongTypeException wte) {
             wte.printStackTrace();
         }
      }
         // The agent agrees to believe of information one the temperature only
         // if the agent which sends it is has sensor.
       return
         (((DisplayAgent)getAgent()).selectedAgent != null
            && agent.equals(((DisplayAgent)getAgent()).selectedAgent))
          || !((SLPatternManip.match(temperatureDefinition.VALUE_X_PATTERN, toBeTested) != null)
            ||  (SLPatternManip.match(temperatureDefinition.NOT_VALUE_X_PATTERN,  toBeTested)  !=
null)
             ||  (SLPatternManip.match(temperatureDefinition.VALUE_GT_X_PATTERN,  toBeTested)  !=
null)
             || (SLPatternManip.match(temperatureDefinition.NOT_VALUE_GT_X_PATTERN, toBeTested)
!= null));
```

3.3 Filter

A new assert filter is added to the display agent in order to display the temperature when a new value is asserted in the knowledge base. This is realized by defining a `KBAssertFilterAdapter` with a specific `afterAssert` method (see the `DisplayCapabilities` class).

```
((KBFilterManagment)myKBase).addKBAssertFilter(
  new KBAssertFilterAdapter("(B ??agent " + temperatureDefinition.VALUE_X_PATTERN + ")") {
     public void afterAssert(Formula formula) {
        try {
         ((DisplayAgent)myAgent).display.setTemperature(((RealConstantNode)applyResult.getTerm("?
?X")).lx_value());
        } catch (ClassCastException cce) {
             try {
               ((DisplayAgent)myAgent).display.setTemperature(((IntegerConstantNode)applyResult.g
etTerm("??X")).lx_value().doubleValue());
             } catch (Exception e) {
                 e.printStackTrace();
             }
       } catch (Exception e) {
           e.printStackTrace();
       }  }});
```

4 Son agent

The son agent is the most complex semantic agent of the demonstration.

4.1 *Defining the son behaviour*

The `EventCreationObserver` makes it possible to trigger internal events each time a given fact becomes true in the knowledge base of the agent (this mechanism is widely used within the JADE Semantics framework to handle the subscribe, request-whenever and request-when messages). The Son Agent has height such observers, which define his behaviour for putting on or taking off clothing items. These height observers are all written on the same pattern. The first formula represents the triggering semantic event, i.e. the formula that becomes true, and the second one represents the semantic event that must be generated when the observer is applied.

Example:

```
getMyKBase().addObserver(new EventCreationObserver(myAgent,
   SLPatternManip.fromFormula("(B "+getAgentName()+" (temperature_gt 20))"),
   SLPatternManip.fromFormula(
      "(and (I "+getAgentName()+" (not (wearing "+getAgentName()+" trousers)))" +
      "(and (I "+getAgentName()+" (not (wearing "+getAgentName()+" pullover)))" +
      "(and (I "+getAgentName()+" (not (wearing "+getAgentName()+" coat)))" +
      "(I "+getAgentName()+" (not (wearing "+getAgentName()+" cap))))))" )));
```

Here, the first formula means: “The agent believes that the temperature is higher than 20 degrees”. The second one means: “The agent has the intention to be dressed without his trousers, his pullover, his coat, and his cap”. The content of each intention corresponds to the rational effect of the `TAKE-OFF` ontological action, so that, when the corresponding semantic event is triggered, the agent will perform the `TAKE-OFF` action in accordance with the rationality principle.

4.2 *Intention transfer*

The Son Agent believes all the other agents. So, unlike the sensor and the display agents, the `acceptBeliefTransfer` method of the son agent does not need to be overridden. However, the son agent should not adopt the intentions of all other agents, instead, it must obey only the mother agent. In this purpose, the `acceptIntentionTranfer` is overridden (see the `ManCapabilities` class) in a similar way to the `acceptBeliefTransfer` method.

```
setMyStandardCustomization(new StandardCustomizationAdapter() {
            // This method prevents the agent from adopting any intention from any agent
            // but the mother agent
      public boolean acceptIntentionTransfer(Formula goal, Term agent) {
          try {
                      return   agent.equals(SLPatternManip.instantiate(AGENT_TERM,   "agent",   new
WordConstantNode(motherAID.getName())));
                  }
                  catch (Exception e) {return false;}}    }));
```

4.3 Ontological Actions

The Son Agent is able to deal with three ontological actions: `PUT-ON` (putting on a clothing item), `TAKE-OFF` (taking off a clothing item), `WAIT` (waiting a while). The `OntologicalAction` class enables to define such actions semantically. The constructor of this class expects three parameters:

- the first one is an SL pattern defining the action expression of the ontological action (meta variables should be used to represent parameters of the action),
- the second one is an SL pattern defining the post-condition of the action (the same meta variables can be used to refer parameters of the action)
- the third one is an SL pattern defining the feasibility precondition of the action (the same meta variables can be used to refer parameters of the action).

Moreover, the method `perform` can be overridden to define the behaviour of the action. This method must be written as the `action` method of a Jade behaviour.

Here is the code of the `PUT-ON`, the `TAKE-OFF` and the `WAIT` actions:

```
getMySemanticActionTable().addSemanticAction(new OntologicalAction(getMySemanticActionTable(),
    "(PUT-ON :clothing ??clothing)",
    SLPatternManip.fromFormula("(wearing ??sender ??clothing)"),
    SLPatternManip.fromFormula("(not (wearing ??sender ??clothing))")) {
            public void perform(OntoActionBehaviour behaviour) {
                ((ManAgent)myAgent).putOn(getActionParameter("clothing").toString());
                behaviour.setState(SemanticBehaviour.SUCCESS);
            }
});

getMySemanticActionTable().addSemanticAction(new OntologicalAction(getMySemanticActionTable(),
    "(TAKE-OFF :clothing ??clothing)",
    SLPatternManip.fromFormula("(not (wearing ??sender ??clothing))"),
    SLPatternManip.fromFormula("(wearing ??sender ??clothing)")) {
            public void perform(OntoActionBehaviour behaviour) {
                ((ManAgent)myAgent).takeOff(getActionParameter("clothing").toString());
                behaviour.setState(SemanticBehaviour.SUCCESS);
            }
});

getMySemanticActionTable().addSemanticAction(new OntologicalAction(getMySemanticActionTable(),
    "(WAIT :time ??time)",
    SLPatternManip.fromFormula("true"),
    SLPatternManip.fromFormula("true")) {
          private long wakeupTime = -1, blockTime;

          public void perform(OntoActionBehaviour behaviour) {
              switch (behaviour.getState()) {
              case SemanticBehaviour.START: {
                   if (wakeupTime == -1) {
```

```
                                                                  wakeupTime    =
System.currentTimeMillis()+Long.parseLong((getActionParameter("time").toString()));
                        }
                    // in this state the behaviour blocks itself
                    blockTime = wakeupTime - System.currentTimeMillis();
                    if (blockTime > 0)
                      behaviour.block(blockTime);
             // The value here is a non predefined value that means that the state of is
             // not START (the exact value is not important).
                    behaviour.setState(1000);
                    break;
                  }
                  case 1000: {
                    // in this state the behaviour can be restarted for two reasons
                    // 1. the timeout is elapsed and the behaviour is definitively
                    // finished
                    // 2. a message has arrived for this agent then it blocks again
                    blockTime = wakeupTime - System.currentTimeMillis();
                    if (blockTime <= 0) {
                      // timeout is expired
                      behaviour.setState(SemanticBehaviour.SUCCESS);
                    } else
                      behaviour.block(blockTime);
                    break;
                  }
                  default : {
                    behaviour.setState(SemanticBehaviour.EXECUTION_FAILURE);
                    break;
                  }
                  } // end of switch
});
```

An `OntologicalAction` automatically checks the action precondition and asserts the action post-condition when the action is performed by the agent. The body (defined in the `perform` method) is only the specific code of the action. For example, the `PUT-ON` action repaints the son panel.

4.4 Initial knowledge

The initial knowledge of the son agent (he wears no clothing item) is set up by the following trivial code (note the extensive use of Patterns throughout the code of each semantic agent):

```
SLPattern initialKPattern =
    SLPatternManip.fromFormula("(B "+  getAgentName()  +" (not (wearing "+ getAgentName() +" ??clothing)))");
getSemanticInterpreterBehaviour().interpret(((Formula)SLPatternManip
        .instantiate(initialKPattern, "clothing", new WordConstantNode("cap")));
getSemanticInterpreterBehaviour().interpret((Formula)SLPatternManip
        .instantiate(initialKPattern,"clothing", new WordConstantNode("coat")));
getSemanticInterpreterBehaviour().interpret((Formula)SLPatternManip
        .instantiate(initialKPattern,"clothing", new WordConstantNode("trousers")));
getSemanticInterpreterBehaviour().interpret((Formula)SLPatternManip
        .instantiate(initialKPattern,"clothing", new WordConstantNode("pullover")));
```

The following code:

```
((FilterKBaseImpl)getMyKBase()).addClosedPredicate(SLPatternManip.fromFormula("(wearing
"+getAgentName()+" ??c)"));
```

is used to inform the agent that it knows all the values that makes true the predicate `wearing`.