france telecom

www.francetelecom.com/

(Unrestricted)

Research & Development

Jade Semantics Add-on Programmer's guide

Author: Vincent PAUTRET

Version: 1.0

France Telecom
Research & Development

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

Table of Contents

This guide assumes the reader to be familiar with the FIPA standards, at least with the *Agent Management* specifications (FIPA no. 23), the *Agent Communication Language*, and the *ACL Message Structure* (FIPA no. 61). The majority of code examples are extracted from the temperature application demonstration.

1 Introduction

This add-on aims at taking better benefit from the semantic dimension of the FIPA-ACL language, which is currently not explicitly taken into account in the JADE platform.

At a basic level, this new framework checks syntactic and semantic consistency of received and sent messages (for example, it is not possible to send an Inform message with a content representing an action instead of a proposition). At a higher level, "semantic agents" developed upon this framework automatically handle each incoming message and react according to its formal semantics as specified by the FIPA standard. For example, no explicit programming is required to handle a Query-Ref, a Call-For-Proposal or a Subscribe message. To this end, a proper API has been defined to customize the behaviour of semantic agents. Roughly speaking, developing a semantic agent mainly consists in setting up its initial beliefs, the rules for handling its beliefs, the domain-specific actions it is expected to handle, and in customizing its generic interpretation behaviour (such as its cooperative abilities).

We hope this new way of programming JADE agents will simplify their coding by relieving developers of boring and recurrent tasks such as parsing and analysing incoming messages. We also envision an increased interoperability between the resulting semantic agents because of their conformance to the actual meaning of the exchanged messages.

1.1 Concepts

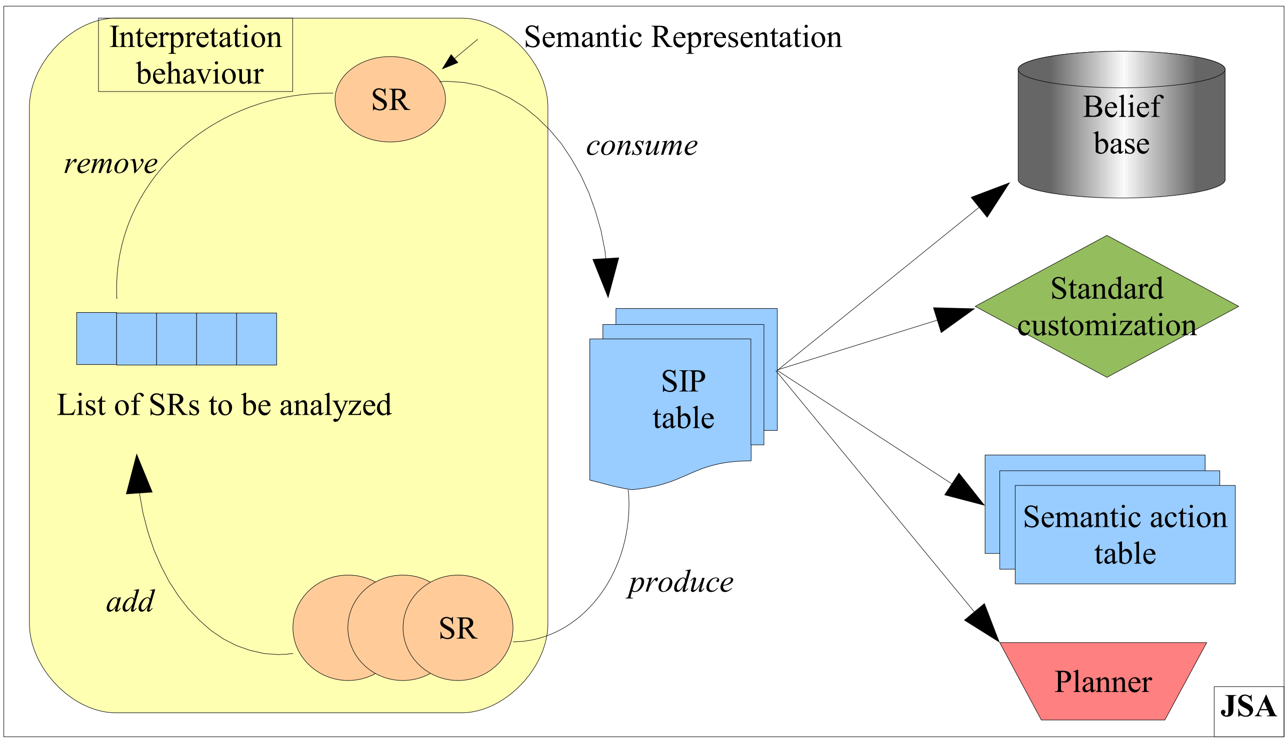


Figure 1: Semantics framework components

A Jade Semantic Agent (JSA) is a JADE agent build upon the Jade Semantics framework. The framework consists of seven main components:

- **Semantic Representations** (or SR) are FIPA-SL expressions that represent the sense of the message according to the FIPA-ACL semantics (see paragraph 4). For example, the formula:

```
(B ja (done
          (action s
                    (CONFIRM  :sender s
                              :receiver ja
                              :content "(p)"))
          true))
```

  stating the receiver (`ja`) believes the sender (`s`) has just issued the act CONFIRM;

- The **belief base** is another component that stores the beliefs of the agent. A belief can be a simple predicate, or a more complex formula;

- The **semantic action table** stores the description (pre- and post-condition) and also the code of all actions the agent can deal with. Each action is associated to a semantic behaviour that performs the action;

- The **standard customization object** is the means the programmer uses to customize the interpretation behaviour in respect to belief adoption, intention transfer, and so on;

- The **planner** is a component used to find a plan that reaches a given goal;

- The **Semantic Interpretation Principles table**. This table holds the basic principles for interpreting messages according to the FIPA-ACL semantics. Each principle (or SIP) may handle SR of a particular form, produce other SRs, add or remove behaviours, or update the belief base. For example, the **ActionFeatures** SIP produces SR representing the semantic features of a received act (mainly rational effect and feasibility precondition);

- Last, the **interpretation behaviour,** which performs the interpretation algorithm, consists in producing and consuming Semantic Representation.

1.2 Semantic interpretation algorithm

The interpretation algorithm is implemented by an object of the class `SemanticInterpreterBehaviour` (extends `jade.core.behaviours.CyclicBehaviour`). It applies on messages, but also on internal events the form of which are SL expressions. It is a loop, which applies all possible ordered SIPs (stored in the SIP table) to all existing SR according to the SIP index of the SR, and stops when one of the following conditions holds:

- the SR list becomes empty,
- no SIP can be further applied to any SR, (in this case the remaining SR are asserted in the belief base)
- one of the SR is a false formula (e.g., in the case of inconsistency). In this case, a Not Understood message is sent.

1.3 Packages overview

The semantics add-on is composed of the following packages:

`jade.semantics:`	parent package of the semantics add-on.
`├ .actions:`	contains the definition of semantic actions and a set of classes to manage them.
`│  ├ .operators:`	contains the classes that define the alternative and sequence of actions operators.
`│  └ .performatives:`	contains all the FIPA communicative acts except the `proxy` and the `propagate` acts, which will be added in a future version.
`├ .behaviours:`	package that provides classes to define and manage semantic behaviours, and some standard behaviours.
`├ .interpreter:`	this package implements the kernel of the semantics add-on.
`│  └ .sips:`	contains all the semantic interpretation principles needed by a JSA to interpret correctly the informations it receives.
`├ .kbase:`	contains a set of generic classes to implement a belief base for the JSA. It provides an implementation of this kind of base based on filters.
`│  ├ .filters:`	sub-package that provides filters definition. Filters are specifics to the provided belief base implementation.
`│  │  ├ .assertion:`	this sub-package provides classes needed to define assertion filters.
`│  │  ├ .query:`	this sub-package provides classes needed to define query filters.
`│  └ .observers:`	this sub-package contains observers which are used to supervise the changes in a belief base.
`├ .lang.sl:`	package is provided to handle SL Language according to FIPA standard specifications.
`│  ├ .content:`	this package provides classes to handle the contents.
`│  ├ .grammar:`	contains all the classes implementing the abstract syntax and the associated basic mechanism.
`│  │  └ .operations:`	this sub-package contains all the operations enable on the grammar.
`│  ├ .parser:`	this package contains classes that implement a parser and an unparser for the SL concrete syntax.
`│  └ .tools:`	this package provides useful tools to handle SL formulae.
`└ .planner:`	contains a Java interface that all planners should implement in order to be readily integrated with the semantics add-on.

This guide is structured as follow. Chapter 2 presents JSA and the way to create it. Chapter 3 details the FIPA-SL language and the associated tools. Then we describe the components of the framework: the Semantic Representation (chapter 4), the belief base and the implementation, which is provided within the framework (chapter 5), the semantic actions (chapter 6), the rules to customize a JSA (chapter 7), a brief overview of planner (chapter 8). and finally the semantic interpretation principles (chapter 9). Chapter 10 gives some rules to use other ontologies and content languages. The last chapter presents useful classes.

2 Jade Semantic Agent

This part describes the way to create a JSA and the semantics add-on classes that support the development of a JSA.

2.1 The SemanticAgent interface and the SemanticAgentBase class

The `SemanticAgent` interface contains the minimal methods which an agent must implement to be a JSA. These methods are:

- the `getSemanticCapabilities()` method which returns the `SemanticCapabilities` object associated to the semantic agent;
- the `getAgent()` method which returns the JADE agent corresponding to this semantic agent.

The `SemanticAgentBase` class represents a common base class for user defined a JSA. Therefore, from the programmer's point of view, a JSA is an instance of a user defined Java class that implements the interface `SemanticAgent`, or more simply that extends the `SemanticAgentBase` class.

Then, the simplest semantic agent is:

```
public class MyJSA extends SemanticAgentBase {}
```

Even if such an agent seems to be empty:

- It automatically answers invalid messages with a not-understood message;
- It can, for example, interpret the content of an inform message, and then answers queries about this content. For example, if I send an inform to this agent the content of which is (temperature 10), and then query about the temperature, the agent answers (temperature 10). However, if I send another inform to this agent with the content (temperature 11), and then query about the temperature the answer may be (temperature 10), because this agent does not correctly handle the temperature predicate. We will see later how such a predicate can be handled consistently.
- Other acts like request-when, subscribe, and so on, are natively handled.

2.2 The SemanticCapabilities class

More generally, a JSA requires mainly a `SemanticCapabilities` object which defines domain specific actions to be handled by the agent, the implementation of the belief base of the agent, including rules to handle domain specific beliefs (for example, the temperature predicate), and the `StandardCustomization` object used by the agent to customize the SIPs.

The `SemanticAgentBase` class uses a default `SemanticCapabilities` object providing the minimum (the default) capabilities:

- the default semantic action table: an `jade.leap.ArrayList` filled with FIPA-ACL communicative acts;
- the default semantic interpretation principles table: a `jade.leap.ArrayList` filled with the predefined list of SIPs;
- the belief base is an instance of `FilterKBase;`
- the standard customization object is an instance of `StandardCustomizationAdapter`, which does nothing;
- no planner.

The programmer has to define his own `SemanticCapabilities` class to take account his/her application features. For this purpose, the `SemanticCapabilities` class provides convenient methods to customize all the components:

- The `setupSemanticActions()` method sets the semantic actions. It creates the semantic action table and loads the communicative actions;
- The `setupSemanticInterpretationPrinciples()` method sets the semantic interpretation principles. It creates the semantic interpretation principle table and loads the principles into the table;
- The `setupKbase()` method sets the belief base. By default, the belief base is a `FilterKbase` instance.
- The `setupPlanner()` method sets the planner. By default, no planner is provided by the semantics framework;
- The `setupStandardCustomization()` method customizes the usage of SIPs.

The following code shows a JSA extending `SemanticAgentBase` class, which redefined a specific `SemanticCapabilities` class.

```
public class MyJSA extends SemanticAgentBase {
        class MySemanticCapabilities extends SemanticCapabilities {
                protected void setupSemanticActions() {...}
                protected void setupKbase() {...}
                protected void setupStandardCustomization() {...}
                ...
        }
        public MyJSA() {

                semanticCapabilities = new MySemanticCapabilities();
        }
        public void setup() {
                super.setup();
                ...
        }
}
```

Like for a classical JADE agent, the `setup()` method of a JSA is the point where any application-defined agent activity starts. The programmer can redefine the `setup()` method as for a classical JADE agent, and **should call the super method** which properly installs the semantic capabilities. In this case, s/he can simply override the different `setupXXX` methods, depending of her/his needs. The manner of defining these methods is described in the sections corresponding to each component.

The `SemanticCapabilities` class provides methods to access all the components of a `SemanticAgent`: belief base, planner, semantic interpretation table, semantic action table, user customisation (`StandardCustomization` class), and the semantic interpreter behaviour:

- The `getAgent()` method returns the semantic agent to which the semantic capabilities object is associated;
- The `getAgentName()` returns the agent name to which the semantic capabilities object is associated;
- The `getMyKBase()` method returns the belief base of the agent;
- The `getMyPlanner()` method returns the planner;
- The `getMySemanticActionTable()` method returns the semantic action table;
- The `getSemanticInterpreterBehaviour()` method returns the semantic interpreter behaviour associated to the agent;
- The `getMySemanticInterpretationTable()` method returns the semantic interpretation principles table;
- The `getMyStandardCustomization()` method returns the standard customisation object.

It is the way to access all the parts of a JSA.

The `SemanticCapabilities` class provides methods to simply send FIPA-ACL messages. There are methods to create each kind of message, and one method to send the created message. The parameters of the methods are the ones used in the message.

- The `sendCommunicativeAction(CommunicativeAction)` method sends the specified communicative action.

For all the methods described below, the last parameter correspond to the list of the receivers of the act. Two prototypes of method exist: the first one with only one Term as last parameter, and the second one with an array of Terms as last parameter. Only the first one is presented in this document.

- The `createAcceptProposal(ActionExpression, Formula, Term)` method creates an `AcceptProposal` act. The first parameter is the action expression denoting the action to be done, and the second parameter gives the conditions of the agreement;
- The `createAgree(ActionExpression, Formula, Term)` method creates an Agree act. The first parameter is the action expression denoting the action to be done, and the second parameter gives the conditions of the agreement;
- The `createCancel(ActionExpression, Term)` method creates a Cancel act. The first parameter is the action expression denoting the action the first agent no longer has the

intention that second agent perform it;

- The `createCFP(ActionExpression, IdentifyingExpression, Term)` method creates a Call For Proposal act. The first parameter is the action expression denoting the action to be done, and the second parameter a referential expression defining a single-parameter proposition which give the preconditions of the action;
- The `createConfirm(Formula, Term)` method creates a Confirm act. The first parameter is the confirmed proposition;
- The `createDisconfirm(Formula, Term)` method creates a Disconfirm act. The first parameter is the disconfirmed proposition;
- The `createFailure(ActionExpression, Formula, Term)` method creates a Failure act. The first parameter is an action expression denoting the failed action, and the second parameter is a formula denoting the reason of the failure;
- The `createInform(Formula, Term)` method creates an Inform act. The first parameter is a formula denoting the proposition the agent wants to inform the other agent;
- The `createNotUnderstood(ActionExpression, Formula, Term)` method creates a Not Understood act. The first parameter is an action expression denoting an not understood action, and the second parameter is an explanatory reason;
- The `createPropose(ActionExpression, Formula, Term)` method creates a Propose act. The first parameter is an action expression denoting the action that the sender is proposing to perform, and the second parameter is a proposition representing the preconditions of the performance of the action;
- The `createQueryIf(Formula, Term)` method creates a QueryIf act. The first parameter is the proposition the agent wants to know if it is true or not;
- The `createQueryRef(IdentifyingExpression, Term)` method creates a QueryRef act. The first parameter is the description of the queried object;
- The `createRefuse(ActionExpression, Formula, Term)` method creates a Refuse act. The first parameter is an action expression denoting the action refused to be perform, and the second parameter is the reason for the refusal;
- The `createRejectProposal(ActionExpression, Formula, Formula, Term)` method creates a Reject Proposal act. The first parameter and the second one consist of an action description and a proposition which formed the original proposal being rejected;
- The `createRequest(ActionExpression, Term)` method creates a Request act. The first parameter is the action to perform;
- The `createRequestWhen(ActionExpression, Formula, Term)` method creates a Request-When act. The first parameter is the action to be perform by the receiver when a proposition (the second parameter) becomes true;
- The `createRequestWhenever(ActionExpression, Formula, Term)` method creates a Request Whenever act. The first parameter is the action to be perform by the receiver when a proposition (the second parameter) becomes true and thereafter each time the proposition becomes true again;
- The `createSubscribe(IdentifyingExpression, Term)` method creates a Subscribe act.

The first parameter is the referential expression identifying an observed object.

The three following methods make it possible to create actions `Unsubscribe`. These actions are not present in FIPA specifications. They make it possible to finish properly the entire effect of a `Subscribe` communicative act.

- The `createUnsubscribe(IdentifyingExpression, Term)` method creates an `Unsubscribe` act. The first parameter is the referential expression identifying the observed object. This method is used with the pattern *((not (I ??agent (done (action ??receiver (INFORM-REF :sender ??receiver :receiver (set ??agent) :content ??ire))))))* where *??ire* is replace by the first parameter of the method;
- The `createUnsubscribe(ActionExpression, Term)` method creates an `Unsubscribe` act. This method is used with the pattern *((not (I ??agent (done ??action))))*. The *??action* metavariable is replaced by the first parameter of the method;
- The `createUnsubscribe(ActionExpression, Formula, Term ...)` method creates an `Unsubscribe` act. This method is used with the pattern *((or (forall ?e (not (done ?e (not (B ??receiver ??property))))) (or (not (B ??receiver ??property)) (not (I ??agent (done ?? action))))))*. The metavariable *??action* is replaced by the first parameter of the method, and the metavariable *??property* is replaced by the second parameter of the method.

Of course, it is not possible to create an `InformIf` or an `InformRef` message. For example, in the case of the `InformIf` message, it has no sense to inform another agent whether or not a given proposition is believed.

The following code shows an example of sending a `RejectProposal` message using these methods.

```
getSemanticCapabilities().sendCommunicativeAction(
      getSemanticCapabilities().createRejectProposal(action,
                                                     selectedCondition,
                                                     new TrueNode(),
                                                     selectedAgent));
```

2.3 Adding semantic capabilities to an existing Jade agent

An existing classical JADE agent can be transformed into a JSA by:

- implementing the `SemanticAgent` interface, and so
- installing semantic capabilities in it.

But this transformation is not easy and the programmer must pay attention to various points:

- The programmer must identify which messages should not be catched by the JSA. If certain messages miss, certain reactions will not be inevitably coherent with the global behaviour (memory of the facts). In particular, if Inform messages are not taken into account, some information will not be assert in the base.
- A semantic agent uses a belief base. If the classical jade agent has already a belief base, it could be necessarily to make a link between these bases.
- At least, the ontological actions defined within the semantic agent framework can not be used by the classical agent.

3 FIPA SL language handling

3.1 Grammar

FIPA-SL expression is the main data type, which the semantic interpretation mechanism relies on. The semantics framework is supported by an internal “Node” structure based on an abstract grammar directly defined from the FIPA-SL concrete grammar specification (see *FIPA SL Content Language Specification*). Consequently, this internal representation exactly matches the SL grammar and each element of the SL grammar corresponds to a unique Java class. The complete SL grammar is shown in Appendix. To date, semantics agents doesn't use at all the Abs structure provided by the JADE framework, but a bridge between Abs and SL nodes is planed for the next version of the add-on.

The next figure shows a part of the abstract grammar related to formula.

```
FORMULA ::= ATOMIC_FORMULA
      | UNARY_LOGICAL_FORMULA
      | MODAL_LOGIC_FORMULA
      | ACTION_FORMULA
      | QUANTIFIED_FORMULA
      | BINARY_LOGICAL_FORMULA
      | meta_formula_reference;

ATOMIC_FORMULA ::= proposition_symbol
                | result
                | predicate
                | true
                | false
                | equals;

MODAL_LOGIC_FORMULA ::= belief
                     |  uncertainty
                     |  intention
                     |  persistent_goal;
```

Figure 2: Part of the SL grammar

SL expressions are implemented as Directed Acyclic Graphs. Each node of these graphs represents an element of the FIPA-SL grammar specification. For example, the formula *(I John (B Bob (temperature 10)))*, means that John has the intention that Bob knows that the temperature is ten degrees[1]. Its representation in the form of graph is shown on figure 3.

1 We distinguish three **mental attitudes**:
Belief, represented by the “B” operator. *(B agt phi)* means “it is true that agent *agt* believes the expression *phi* is true”;
Intention, represented by the “I” operator. *(I agt phi)* means “it is true that agent *agt* intends that expression *phi* becomes true and will plan to bring it about”.
Uncertainty, represented by the “U” operator. *(U agt phi)* means “it is true that agent *agt* is uncertain of the truth of *phi*”.

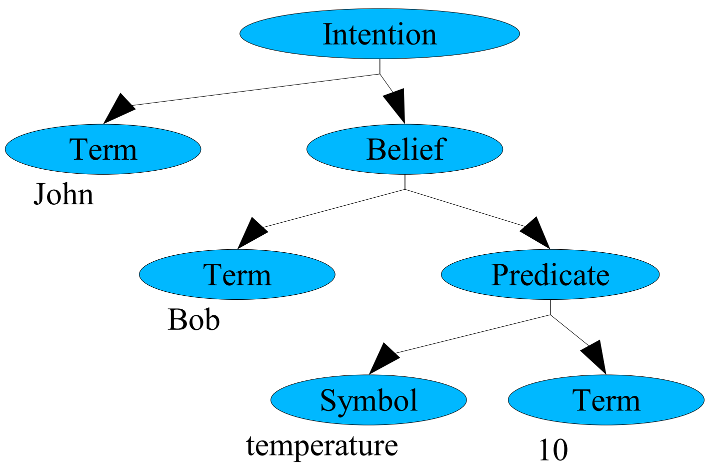


Figure 3: Graphical representation of a formula

It is possible to define additional operations on each kind of node of the SL grammar. Depending on the needs in SL expression handling, this set of operations can be easily extended. To date, the most important ones are:

- for the Formula node:
  - The `getsimplifiedFormula()` method computes the normal form (which is logically equivalent according to the SL semantics) of any SL formula node (this point is more discussed in section 3.2). For example: `SLPatternManip.fromFormula("(B agt (I agt p))")`**`.getSimplified()`** returns a SL formula node representing the `(I agt p)` formula.
  - The `isSubsumedBy(Formula)` method checks if the considered SL formula node is subsumed by a given formula (according to the SL semantics). For example: `SLPatternManip.fromformula("(not (B agt p))")`**`.isSubsumedBy`**`(SLPatternManip.fromFormula("(I agt p)"))` returns `true`.
  - The `isMentalAttitude(Term)` method checks if the considered SL formula node is a mental attitude (i.e. Belief, intention, or uncertainty) of the given agent. For example, `SLPatternManip.fromformula("(B (agent-identifier :name foo) p)")`**`.isMentalAttitude`**`(SLPatternanip.fromTerm("(agent-identifier :name foo)"))` returns `true`.
  - The `isBeliefFrom(Term)` method checks if the considered SL formula node is a belief of the given agent. For example, `SLPatternManip.fromformula("(B (agent-identifier :name foo) p)")`**`.isBeliefFrom`**`(SLPatternanip.fromTerm("(agent-identifier :name foo)"))` returns `true`.

- for the Term node:
  - The `getSimplifiedTerm()` method has the same effect on the terms as the preceding one on formulae.

- for the Content node:
  - The `getContentElement(int)` method returns the element of the content at the given index;
  - The `setContentElement(int, Node)` method sets the element at the given index with the given node;

- The `addContentElement(Node)` method adds a new element in the content;
- The `setContentElements(int)` method creates a content element storage of the given size, or, if it already exists, clears it;
- The `contentElementNumber()` method the number of element in the content.

- for all nodes:
  - The `toString()` method unparses any node in a String representation format.

Finally, meta-operators are included into the SL grammar to improve the pattern expressiveness. To date, only one meta-operator is defined:

- This operator (identified with the characters "::?") handle optional sub-nodes in an SL element.

Several additional meta-operators are planned for future versions.

3.2 Normal form of SL expression

The normal form of SL formulae is used in order to provide unambiguous semantic representation of messages and unambiguous applications of SIPs. We define a set of rules to compute a normal form (it is theoretically not possible to compute a real normal form that is a unique formula representing all the logically equivalent formulae).This set seems to be sufficient for most of cases. However, it can be completed in order to get more accurate resulting normal forms. For example, the set contains the rules:

- *(not true)* is simplified as *false*;
- *(not (not phi))* is simplified as *phi*;
- *(not (forall ?x phi))* is simplified as *(exists ?x (not phi))*;
- *(B ja phi)* is simplified as *phi*, if *phi* is a mental attitude of *ja*;
- *(B ja (and phi psi))* is simplified as *(and (B ja phi) (B ja psi))*;
- etc.

Also, we arbitrary decided to organise parameters in an alphabetic order in order to makes the pattern matching (see next paragraph) process easily. For example, the normal form of the term:

```
(action (agent-identifier :name ja) (INFORM :sender (agent-identifier :name ja)
:receiver  (set (agent-identifier :name ja1)):content "((temperature 10))"))
```

is:

```
(action (agent-identifier :name ja) (INFORM :content "((temperature 10))"  :receiver
(set (agent-identifier :name ja1)) :sender (agent-identifier :name ja))
```

The simplification process is accessible by the use of the `getSimpliedXXX` methods associated to formulae, terms and contents.

3.3 Pattern matching

3.3.1 Generality

The semantics framework provides useful tools to easily handle SL elements (formulas, terms,

contents and more generally any kind of node of the SL grammar).
The `SLPatternManip` class makes it possible to parse/unparse various SL formulas and terms (in addition to contents) from/to Java Strings.

- The `fromFormula(String)` method returns the formula node corresponding to the given expression (returns null if the expression is not a formula according to the SL grammar);
- The `fromTerm(String)` method returns the term node corresponding to the given expression (returns null if the expression is not a term according to the SL grammar);
- The `fromContent(String)` method returns the content node corresponding to the given expression (returns null if the expression is not a content according to the SL grammar);

For example:

```
Formula f = SLPatternmanip.fromFormula("(not (temperature 10))");
Term term = SLPatternManip.fromTerm("(agent-identifier :name foo)");
Content content = SLPatternManip.fromContent("((temperature 10))");
```

More interestingly, it additionally provides a powerful mechanism of pattern handling for creating, recognizing, and analysing any SL elements. This mechanism relies on the introduction of "meta-variables" into the SL grammar (identified with a double "?" character), which can be matched with any regular SL element (depending of the context). When matching two SL expressions, the instantiated values of these special elements can be easily retrieved (the results are stored in an instance of `MatchResult` class[1]).

- The `match(Node, Node)` method checks if the two expressions match. If true, all meta variables that has been unified can be accessed using one of the `getFormula(String)`, `getTerm(String)`, `getVariable(String)`, `getSymbol(String)`, or `getContentExpression(String)` methods of the matching result (`MatchResult` class). This method is equivalent to `SLPatternManip.newMatcher().match(expression1, expression2)`;

Of course, several variables may occur within the SL pattern. On the contrary, if the same variable occurs several times within a SL pattern, each occurrence should correspond to an identical part of the unified SL expression.
For example:

- Creating a pattern

```
Formula pattern = SLPatternManip.fromFormula("(B ??agent ??phi)");
```

- Instantiating a pattern

```
Term agent = SLPatternManip.fromTerm("(agent-identifier :name foo)");
Formula phi = SLPatternManip.fromFormula("(temperature 10)");
Formula formula = SLPatternManip.instantiate(pattern,
                                                    "agent", agent,
                                                    "phi", phi);
System.out.println(formula);
```

1 A MatchResult is a list of metavariables linked to their values.

echoes: `(B (agent-identifier :name foo) (temperature 10))`

- Matching patterns:

```
Formula myPattern = (B ??agent (wearing ??agent coat));
Formula myFormula1 = (B (agent-identifier :name foo)
                     (wearing (agent-identifier :name foo) coat);
Formula myFormula2 = (B (agent-identifier :name foo)
                     (wearing (agent-identifier :name foo) cap);
Formula myFormula3 = (B (agent-identifier :name foo)
                     (wearing (agent-identifier :name foo2) coat);

MatchResult result = SLPatternManip.match(myPattern, myFormula1);
if (result != null)
          System.out.println("agent = " + result.getTerm("agent"));
```

echoes: `agent = (agent-identifier :name foo)`

```
result = SLPatternManip.match(myPattern, myFormula2);
if (result != null)
          System.out.println("agent = " + result.getTerm("agent"));
```

echoes nothing because `myPattern` and `myFormula2` do not match and so the result of the method `match` is `null`.

```
result = SLPatternManip.match(myPattern, myFormula3);
if (result != null)
          System.out.println("agent = " + result.getTerm("agent"));
```

echoes nothing because in `myFormula3` the two agents are different whereas in the pattern `myPattern` the same agent is expected.

The instantiation mechanism consists in giving a value to each metavariable. It can be done from two different ways. Firstly, the simple way, by using one of the `instantiate` methods.

- The `instantiate(Node)` method returns a tree representing the instantiated pattern, meaning all meta-variables have been replaced by their value;
- The `instantiate(Node, String, Node)`, `instantiate(Node, String, Node,String, Node)`, `instantiate(Node, String, Node, String, Node, String, Node)`, and `instantiate(Node, String, Node, String, Node, String, Node, String, Node)` methods return the instantiated pattern (the first parameter) with each metavariable denoted by a `String` replaced by the corresponding `Node` value (these methods call the method `set` described below).

Secondly, by setting step by step the metavariables and then by calling the `substitueMetaReferences` method which substitutes each metavariable by its value:

Warning: In this case, take care to clone the pattern before beginning the instantiation.

```
Formula pattern = SLPatternManip.fromFormula("(B ??agent ??phi)");
Formula phi = SLPatternManip.fromFormula("(temperature 10)");
Formula result = (Formula)pattern.getClone();
SLPatternManip.set(result,
          "agent", getSemanticCapabilities().getAgentName());
SLPatternManip.set(result, "phi", phi);
SLPatternManip.substituteMetaReferences(result);
```

With :

- the `set(Node, String, Node)` method sets in the result pattern the metavariable designed by the String with the Node value given as third parameter;
- the `substituteMetaReferences(Node)` method substitutes in the given pattern all the metavariables with their corresponding values.

AT least, it is interesting to call the `getSimplifiedFormula()` method after an instantiation to be sure that the produced formula is in the normal form. Example, the following code :

```
Formula pattern = SLPatternManip.fromFormula("(not ??phi)");
Formula f1 = SLPatternManip.fromFormula("(not p)");
Formula f2 = (Formula)SLPatternManip.instantiate(pat, "phi", f1);
if (SLPatternManip.match(f2, SLPatternManip.fromFormula("p")) != null) {
        System.out.println("Success !");
} else {
       System.out.println("Fail !");
}
```

echoes: `Fail!`

Otherwise, with the `getSimplifiedMethod()`:

```
Formula f2 = ((Formula)SLPatternManip.instantiate(
             pat, "phi", f1)).getSimplifiedFormula();
```

it echoes: `Success !`

3.3.2 AndNode & OrNode

The pattern matching is order independent for the And Formula and the Or formula. For example,

```
Formula myPattern = (or (s 1) (and (and (p 1) (q 2) (and (r 1) (t 2))));
Formula myFormula1 = (or (and (t 2) (and (and (q 2) (p 1)) (r 1)) (s 1));

MatchResult result = SLPatternManip.match(myPattern, myFormula1);
if (result != null) System.out.println("Success !"));
```

echoes: `Success !`

3.3.3 EqualsNode

The pattern matching take account the commutativity of the equals operator So, the following code:

```
Formula f1 = (= (p 1) (q 2));
Formula f2 = (= (q 2) (p 1));

MatchResult result = SLPatternManip.match(f1, f2);
if (result != null) System.out.println("Success !"));
```

echoes: `Success !`

3.3.4 Optional parameters

Finally, it is possible to precise optional parameters in the pattern using the meta-operator defined in the previous section. For example, the following pattern:

```
Term pattern = SLPatternManip.fromTerm("(agent-identifier " +
                                "(::? :addresses ??addresses) " +
                                ":name ??name " +
                                "(::? :resolvers ??resolvers))");
```

defines a term with two optional parameters: addresses and resolvers. This kind of pattern could match, for example, these terms:

```
Term term1 = SLPatternManip.fromTerm("(agent-identifier :name foo)");
Term term2 = SLPatternManip.fromTerm(
          "(agent-identifier :addresses add :name foo)");
```

3.4 Dynamic building of SL expressions

It is possible to handle directly the abstract tree representing a SL expression by adding new nodes or removing existing ones. The different methods which make it possible to handle the nodes of the tree are shown is the Node hierarchy shown in the Appendix. For example, for the `ModalLogicFormula` node (this kind of formula is in the form of "*(ModalOp agent formula)*"), there are four methods which are getters and setters:

ModalLogicFormula
as_agent as_formula

- The `as_agent()` method returns the agent of the modal logic formula;
- The `as_agent(Term)` method sets the agent of the modal logic formula;
- The `as_formula()` method returns the formula of the modal logic formula;
- The `as_formula(Formula)` method sets the formula of the modal logic formula.

Then, it is easy to create a modal logic formula like this:

```
ModalLogicFormula formula = new BelieveNode();
formula.as_agent( SLPatternManip.fromTerm("(agent-identifier :name        foo)"));
formula.as_formula(SLPatternManip.fromFormula("(temperature 10)"));
System.out.println(formula)
```

echoes: `(B (agent-identifier :name foo) (temperature 10))`

This approach can be interesting in term of performance. The “manual” construction of the tree in particular avoids parsing the strings representing SL expressions.

4 Semantic Representation

Semantic Representations(SRs) are the elements handled by the mechanism of interpretation (see Figure 1: Semantics framework components). A Semantic Representation is a data structure defined by four attributes:

- the ACL message which made it possible to build the SR. If the value of this attribute is `null`, that means that the SR results from an internal event to the agent;
- a SL Formula that represents the sense of the received message or internal event;
- the index of the first semantic interpretation principle it is possible to apply on this SR. By default, the value of the initial index is 0, meaning that all the SIPs will be tested on it;
- the data to feed back. It is mainly used in the intention transfer SIP. This SIP expresses a necessary cooperation principle of a JSA (receiving an ACL message) towards the intentions that the sender intends to communicate. This step is typically used to interpret incoming `Request` or `Inform` messages, the content of which being an intention of the sender (the goal to reach). When a JSA adopts feed back data attribute is set so that the JSA automatically sends a message indicating that it intends to achieve the goal. This attribute is `null` for internal events. The `FeedBackData` class is an inner class of the `SemanticRepresentation` class. It is a data structure defined by two attributes: the receiver of the feed back and the goal to reach.

SRs are created by different ways:

- each message received by a JSA is translated into a SR, which internally represents its meaning;
- SIPs can produce SRs representing new meanings (that may be seen as inferred by the interpretation process), which conform to the FIPA-ACL semantics;
- behaviours can produce SRs to be interpreted as internal event of the agent.

In the last case, the method `interpret` is used. This method is provided by the `SemanticInterpretationBehaviour` class, which is the mean to interpret correctly the given SR. This SR is added to the internal SR list of the behaviour and analyse using the SIPs as an external message. The `SemanticInterpretationBehaviour` provides three methods of this kind:

- the `interpret(SR)` method creates an event (a `SemanticRepresentation`) in the internal event list;
- the `interpret(Formula)` method creates an event (a `SemanticRepresentation`) in the internal event list from the given formula;
- the `interpret(String)` method creates an event (a `SemanticRepresentation`) in the internal event list from the given `String`, representing a formula.

This method restarts the `SemanticInterpreterBehaviour` if it was previously blocked.

In fact, **this method is the way to use when the programmer wants to add formulae in the interpretation mechanism or to add beliefs in the belief base**. So, s/he can be sure that all deductions are done. For example, the following code extracted from an agent *ja* (in its

SemanticCapabilities instance), adds the formula *(wearing ja coat)* in the base.

```
getSemanticInterpreterBehaviour().interpret(((Formula)SLPatternManip
.instantiate(SLPatternManip.fromFormula("(B ja (wearing ja coat))")));
```

5 Belief base

This section describes the usage of the Belief Base component supported by the framework. The base stores facts believed by the agent according to the specific application domain.

Without any programming effort, this component can only store and retrieve "raw" facts without actually understanding (or interpreting) their meaning. For instance, if an empty JSA is informed that `(temperature 10)` is true, then he will correctly answers a Query-Ref message about `(any ?T (temperature ?T))` (meaning "give me a value of the temperature predicate"). However, he will not be able to answer a Query-If message about `(temperature_gt 15)` (meaning "is the value of the temperature predicate greater than 15?"), because he cannot guess the semantic relationships between the `temperature` and the `temperature_gt` predicates.

This section describes the different operations towards the belief base, which are needed by the interpretation process. The belief base should have two characteristics:

- storing the facts believed by the agent;
- providing a mechanism allowing observing variations in the contents of the base.

5.1 Principles

The interface `KBase` defines the methods a JSA belief base should have to manage this beliefs:

- The `assertFormula(Formula)` method asserts a formula in the belief base. It is up to the belief base to manage possible inconsistencies. In any case, each asserted fact should hold in the belief base just after the corresponding asserting operation. For example, assuming that (not p) is in the belief base, p will also be in the base after an assertion of p. Depending on the implementation of the belief base, (not p) may be removed in order to maintain a consistent state of the base;
- The `queryRef(IdentifyingExpression)` method returns a list of objects that satisfy a property belonging to the belief base ("object queries"). In more technical terms, the "object queries" enable to find out an object (or a set of objects) *o* such as *(= (Ref ?X (q ?X)) o)* belongs to the belief base, given a query formula *q* and an SL referential operator Ref (iota, any or all). The three referential operator definitions, which can be found in the FIPA SL Content language specification, are:
    - `iota`, meaning "look for the **unique** object *o* satisfying *p*". If exactly one object *o* can be found such as `(p o)` is true, then the method returns a list including only this object. Otherwise (if there are several or no such objects) the method returns `null`;
    - `any`, meaning "look for any object *o* satisfying *p*". If one object *o* can be found such as `(B jsa (p o))` is true, then the method returns a list including only this object. Otherwise (if there is no such object) the method returns `null`;

- `all`, meaning “look for **all** objects *`o`* satisfying *`p`*”. The method returns a list including all the objects *`o`* that can be found such as `(`*`p o`*`)` is true. If no such object can be found, it returns an empty list, but never returns `null`;

We introduce a new referential operator:

- `some`, meaning “look for some objects *`o`* satisfying *`p`*”. The method returns a list including all the objects *`o`* that can be found such as `(B `*`jsa`*` (`*`p o`*`))` is true. If no such object can be found, it returns an empty list, but never returns `null`.

Correct use of these operators makes it possible to handle precisely the beliefs of the agent. A predicate `p` is considered as closed when an agent knows the value that makes the predicate true, and for all the others value `?y`, `(not (p ?y))` is true.

Let's take an example:

If a JSA is informed that `(= (iota ?x (temperature ?x)) 10)` is true, then its answer to a Query-Ref message about `(iota ?T (temperature ?T))` (meaning “give me the only value of the temperature predicate”) is 10. The answer to a Query-IF message about `(= (iota ?x (p ?x)) 10 )` is true, because the agent knows the only value for the `temperature` predicate.

However, if a JSA is informed that `(temperature 10)` (equivalent to `(= (any ?x (temperature ?x)) 10)`) is true, then he its answer to a Query-Ref message about `(iota ?T (temperature ?T))` (meaning “give me the only value of the temperature predicate”) is “i do not know”. It may exists other values of temperature. On the other hand, its answer to a Query-Ref message about `(iota ?T (B jsa (temperature ?T)))` (meaning “give me the only value you believe of the temperature predicate”) is 10, because the agent knows only one value. And, the answer to a Query-IF message about `(= (iota ?x (p ?x)) 10)` is `null` because it could exists other values and the agent do not know if it is the only value.

When a predicate is considered as closed, it stays closed even if other value are added. For example, if a JSA is informed that `(= (iota ?x (temperature ?x)) 10)` is true (the predicate is considered as closed), and later this agent is informed that `(temperature 11)` is true, its answer to a Query-Ref message about `(all ?T (temperature ?T))` (meaning “give me the all the values of the temperature predicate”) is {10, 11}, because the predicate is always considered as closed. In the temperature demo, the temperature predicate is closed but if there is only one value in the base, it is due to the filters of the base.

It is possible to inform an agent that it is omniscient about a predicate (to inform the agent a predicate is closed) at the set up of the belief base (i.e. in the `setupKbase()` method). For this purpose, the `addClosedPredicate(Formula)` is provided in the given implementation of the belief base (the `FilterKBaseImpl` class).

- The `query(Formula)` method returns a list of solutions to the query on a pattern. If the pattern does not contain any metavariables, an empty list is returned by the method, meaning that the fact belongs to the belief base, and `null` means that the fact does not belong to the belief base. If the pattern contains metavariables, their values are stored in the returned list if the pattern matches a formula (meaning `true`) in the base (to date only one solution is returned), and `null` if the pattern does not match any formula (meaning `false`);
- The `removeFormula(Finder)` method removes from the belief base all formulae recognized

by the finder. This method is deprecated, so prefer the following method:

- The `retractFormula(Formula)` method retracts all the formulae which match the given formula (which could contains metavariables).

The `KBase` interface provides methods to observe the belief base:

- The `addObserver(Observer)` method adds an observer to the belief base at the end of the list of observers;
- The `removeObserver(Finder)` method removes from the belief base all the observers that are identified by the finder.

In a transverse way, the programmer has to:

- identify which information to deal with;
- how this information should be manage.

This will have consequences on the implementation of the belief base.

In our examples, we deal with 3 kinds of information:

- **(temperature x)** predicate is used by all agents. It simply means the temperature value is x;
- **(temperature_gt x)** predicate is also used by all agents. It means the temperature value is greater than x;
- **(wearing agent clothing)** is only used by the son agent and the mother (or daughter) agent (it has no meaning for other agents). It means the agent is wearing a specified clothing.

An important task consists in properly designing this information:

- For example, the **temperature** should be a single value; this should be considered when implementing this predicate
- Another example concerns the **temperature_gt** predicate. This predicate depends on the temperature predicate. Obviously, (temperature_gt x) should return true for all values less than the current temperature value.

Warning:
Remember that it is better to use one of the `interpret()` methods of the class `SemanticInterpretationBehaviour` to add beliefs in the belief base than to use directly the `assertFormula` method (see section 4). By using the interpret method, you are sure that all possible deductions and simplifications are done.

5.2 The Filter belief base

The Filter belief base (`FilterKBase` interface) which is provided within the semantics framework is a very simple base.

The belief base is divided into two lists (`jade.leap.ArrayList`) that contain two kinds of beliefs:

- the beliefs on the facts related to the considered application;
- the beliefs on the realized intentions.

This base provides a simple filter based mechanism to manage the beliefs (see next section) that can be used to add consistency rules or inference rules. Filters are used to manage the accesses to the belief base. It is a way for the developer to trigger some specific code. Two kinds of filters can be used:

- **Assert filters** are automatically called when asserting a fact;
- **Query** filters are automatically called when querying facts or identifying expressions.

The `FilterKBase` interface provides methods to handle filters such as:

- The `addKBAssertFilter(KBAssertFilter)` method adds an assert filter to the belief base;
- The `addKBAssertFilter(KBAssertFilter, int)` method adds an assert filter at the specified index;
- The `removeKBAssertFilter(Finder)` method removes the assert filters that are identified by the specified finder;
- The `addKBQueryFilter(KBQueryFilter)` method adds a query filter to the belief base;
- The `addKBQueryFilter(KBQueryFilter, int)` method adds a query filter at the specified index;
- The `removeKBQueryFilter(Finder)` method removes the query filters that are identified by the specified finder;
- The `addFiltersDefinition(FiltersDefinition)` method adds a list of filters to the belief base (useful for defining specific predicate management).

All these methods are used same manner. For example:

```
((FilterKBase)myKBase).addKBAssertFilter(
            new KBAssertFilter() {...}
}
```

For each kind of filters (assert filters and query filters), a specific **ordered** list (`jade.leap.ArrayList`) is used to store them in the belief base. Two constants are defined in the `FilterKBase` interface to help the addition of the filters in the base:

- `Front` is used to add a filter at the beginning of a list;
- `End` is used to add a filter at the end of the list

5.2.1 Filters

The `KBFilter` class defines the filters. It gives the method to have a link between a filter and the belief base it belongs to by giving the getter (the `getMyKBase` method) and setter (the `setMyKBase` method).

The next paragraphs describe the standard filters provided by the framework.

5.2.1.1 Assert filters

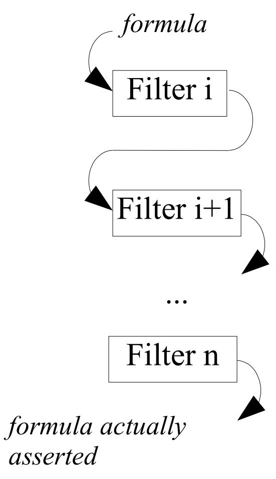
This kind of filter is used to maintain the consistency of the belief base and to manage the storage of the beliefs. The Assert filters of the Temperature Demo illustrate the first point as they maintain the consistency (at each time there is only one value of the temperature in the belief base) of the predicates *temperature* and *temperature_gt* (at each time there is only one value of the temperature in the belief base). The following filter illustrates the second point:

```
// Filter to handle assert of image content.
((FilterKBase)myKBase).addKBAssertFilter(new KBAssertFilterAdapter("(B ??agent
(image-content ??id ??content))") {
    public Formula doApply(Formula formula) {
        String id = ((Constant)applyResult.term("id")).stringValue();
        String location = albumUrl.replaceAll("file:", "")+id+".jpg";
        byte[] bytes = ((ByteConstantNode)applyResult.term("content")).lx_value();
        JPEGUtilities.save(JPEGUtilities.load(new ByteArrayInputStream(bytes)),
location);
        return new TrueNode();
    }
});
```

In this filter, the filter matches a formula where the metavariable ??content is linked to an image content. Instead of storing the image in the belief base of the agent, which would take much place, the image is stored in a file. The filter returns a `TrueNode` formula to prevent that the raw mechanisms assert the image in the base. Of course a query filter is necessary, that loads the image from the file, to get back the image when needed.

Each assert filter could be trigger before and/or after the assertion of a formula in the belief base (the assert filters are defined by the `KBAssertFilter` class):

- The filters are tested at the beginning of the assertion algorithm, in the order where they were added, by calling the `apply(Formula)` method. This method is always called when a formula is asserted and returns a formula, which is the new formula to assert.



  By this way, modifications could be done on the incoming formula if the programmer wants to modify the formula to assert. The programmer can thus bypass the classical assertion mechanism by returning a `TrueNode` formula (a `TrueNode` formula is never asserted in the belief base).

- Then the raw mechanism of the assertion try to assert the formula;

- Finally, the `afterAssert(Formula)` method is called after the assertion for each assert filter in the order defined by the list. The method is called only if a boolean (the attribute `mustApplyAfter` of `KBAssertFilter` class) is set to `true`. However, the `afterAssert(Formula)` method is deprecated, it is better to use an observer (see section 5.2.2).

Here, the default case for these two methods is shown:

```
public Formula apply(Formula formula) {
          // The afterAssert method will be called after the assertion
          mustApplyAfter = true;
          // No modification on the incoming formula
          return formula;
}

// The method does nothing.
public void afterAssert(Formula formula) {
}
```

We present now the standard assert filters provided within the framework.

KBAssertFilterAdapter
This class is an easy means to define an assert filter. The string given in the constructor represents the formula on which the filter could be applied. The programmer must override the `doApply(Formula)` method which is called by the `apply` method if the filter is applicable. S/he can override the `afterAssert` method too, depending of her/his needs.

AndFilter
This filter asserts in the belief base the two parts of an And formula (by splitting the `AndFormula` into two formulae). It is applicable on the formula of the form *(and φ ψ)*. In this case, the method returns a `TrueNode`.

EventMemoryFilter
This filter aims at storing actions or events the agent has done. It is applicable on the asserted formula of the form *(B ja (done act φ))*. If the action expression act is a sequence of actions, all the unitary actions, which make the sequence are stored.
As, the filter use the event memory list to store the actions, the filter stores the formulae itself. The `apply` method returns a `TrueNode` formula and the `afterAssert` method is never called.

ForallFilter
This filter is used to manage the formulae of the form *(forall ??var (B ??agt ??phi))*. If the incoming formula matches this pattern, a formula of the form *(= (all ??var (not ??phi)) (set))* is asserted. In this case, the method returns a `TrueNode`.

IREFilter
This filter is used to manage correctly the expressions of the form *(= (all ??X ??formula) ??set))* or *(= (iota ??X ??formula) ??value))*. Without this filter, it is the whole expression which would be asserted in the belief base. With this filter, the asserted part is:

- for the all expression : each element which appears in the set;

- for the iota expression : the single element.

The filter is applicable on the formulae of the form *(B ??agent (= (all ??X ??formula) ??set))* or *(B ??agent (= (iota ??X ??formula) ??set))*. The `afterAssert` method does nothing.

ObserverFilter
After the assertion of a formula, this filter checks all the observed formulae. If the value of the observation becomes true, the associated observer is notified. The `apply` method does nothing in this filter.

Example of an applicative filter
In our example, we use the `KBAssertFilterAdapter` to remove all previous known fact about the temperature each time a new temperature value is asserted. The method returns the same formula in order to the raw mechanism of the assertion process asserts the formula. However, if the formula is already in the base, the method returns a `TrueNode` formula to avoid that the raw mechanism of the assertion asserts it again.

```
((FilterKBase)myKBase).addKBAssertFilter(
    new KBAssertFilterAdapter("(B ??agent (temperature ??x))") {
   public Formula doApply(Formula formula) {
       if ((myKBase.query(formula) != null)) {
          return new TrueNode();
       } else {
          kbase.retractFormula(SLPatternManip.fromFormula("(temperature ??x)"));
          kbase.retractFormula(SLPatternManip.fromFormula("(not (temperature ??
x))"));
          kbase.retractFormula(SLPatternManip.fromFormula("(temperature_gt ??x)"));
          kbase.retractFormula(SLPatternManip.fromFormula("(not (temperature_gt ??
x))"));
          return formula;
}}});
```

The filter coding is very simple. It takes as an argument the pattern representing the facts to be considered, that is `(B ??agent (temperature ??x))`. If the belief base already holds the asserted fact, the filter returns true and no new fact will be asserted. At the contrary, the filter removes all facts about the temperature; these facts states :

- `(temperature ??x)`
- Or `(not (temperature ??x))`
- Or `(temperature_gt ??x)`
- Or `(not (temperature_gt ??x))`

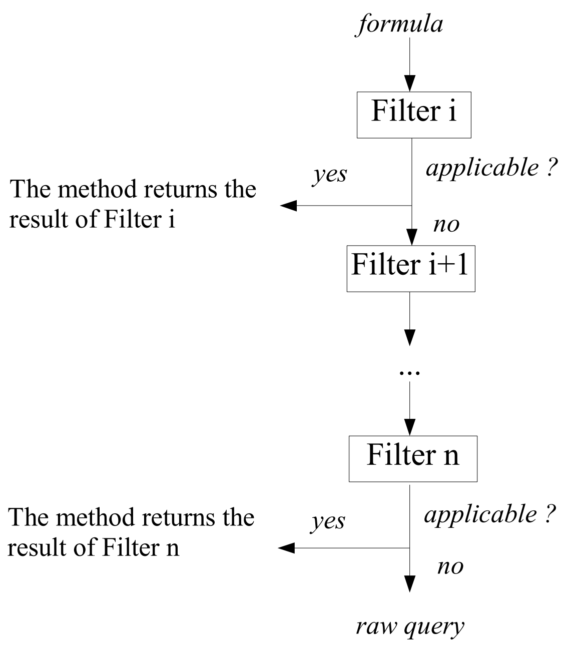
5.2.1.2 Query filters

The `KBQueryFilter` abstract class provides a method to use a query filter:

- The `apply(Formula, Term)` method has two parameters:
    - the queried formula;
    - a term that represents the agent trying to apply the filter;

  and returns a `QueryResult` object which have two attributes:
    - a boolean which is true if the filter is applicable on the formula, false if not;

- a list of MatchResults which contains all the MatchResult, resulting of the performance of the filter on the given formula. Each MatchResult contains a list of value that makes true the formula given in parameter. If the list is null, that means that the answer to the query is false.


formula
Filter i
yes
applicable ?
The method returns the result of Filter i
no
Filter i+1
...
Filter n
The method returns the result of Filter n
yes
applicable ?
no
raw query

All the query filters are called at the beginning of the query algorithm (by calling the apply method) in the order defined by the query filter list. If a filter is applicable (i.e. the incoming formula matches a specific pattern defined in the filter), the algorithm stops and returns the result of this filter. if no filter is applicable, the raw mechanism of the query is applied and the result returned.

We present now the standard query filters provided within the framework.

KBQueryFilterAdapter

This class is an easy means to define a query filter. The string given in the constructor represents the formula on which the filter could be applied. The programmer must override the apply(Formula) method; depending of her/his needs.

The following code shows the apply method of the QueryFilterAdapter.

```
public QueryResult apply(Formula formula, Term agent) {
        // the resut. By default, the boolean is set to false and the list
        // Matchresult is set to null
        QueryResult queryResult = new QueryResult();
        // test if the filter is applicable
        MatchResult match = SLPatternManip.match(formula, pattern);
        if (match != null) {
            // the filter is applicable, the boolean of the result is set to true
            queryResult.setFilterApplied(true);
            ListOfMatchResults list = new ListOfMatchResults();
            // the method doApply returns the list of MatchResult
            list.add(doApply(formula, match));
            queryResult.setResult(list);
        }
        return queryResult;
}
```

In the case of the `QueryFilterAdapter`, the programmer has just to redefine the `doApply` method which returns the list of `MatchResult`.

AndFilter

This filter is applicable when the queried formula is a conjunction (the filter is applicable on the formulae of the form *(and ??phi ??psi)*. The result is the join of each `MatchResult` resulting of the query on *??phi* and all `MatchResults` resulting of the query on *??psi*.

BeliefTransferFilter

This filter is used to check if the semantic agent is cooperative towards another agent regarding a specific belief or not. This filter uses the `acceptBeliefTransfer` method (see paragraph 7) of the `StandardCustomization` class to check if the filter returns true or false. The filter is applicable on formulae of the form *(or (not (I ??agent1 (B ??agent2 ??belief))) (B ??agent2 ??belief))*, meaning *agent1* has the intention *agent2* believes *belief* implies *agent2* believes *belief*. For example, this filter may be applied when the Jade agent receives a `RequestWhenever`, a `Refuse`, `Subscribe`, a `Propose`, a `Confirm`, a `Failure`, a `Disconfirm`, or an `Inform` message with the appropriate content.

CFPFilter

This filter applies whenever an agent (*agent1*) is calling for a proposal (consisting in performing an action (*act*) under a specific condition (*condition*)) towards another agent (*agent2*). This filter is applicable on formulae of the form *(or (not (I ??agent1 (done ??act ??condition))) (I ??agent2 (done ??act ??condition)))*. The apply method, depending of the form of the identifying expression given in parameter, calls the `handleCFPIota` method, the `handleCFPAny` method, the `handleCFPSome` method, or the `handleCFPAll` method of the `StandardCustomization` class (see paragraph 7).

EventMemoryfilter

This filter is applicable on formulae of the form *(B ??agent (exists ??e (done ??act)))***.** This filter checks if the action "*??act*" is a sequence of actions (can be reduced to only one) already done by the agent. If a `VariableNode` appears in the action expression that means an unspecified number of actions can appear between the specified actions. For example, a1;a2;e;a3 , means that a2 must follow a1 in the memory whereas there can be several actions between a2 and a3.

ExistsFilter

This filter is applicable on formulae of the form *(B ??agt (exists ??var ??phi))* or *(exists ??var (B ??agt ??phi))*. The variable *??var* is changed into Metavariable in the formula *??phi* and the method query is called on the resulting formula. The result is the resulting list of `MatchResults` of the last query, in which the result on the variable *??var* has been removed.

ForallFilter

This filter is applicable on formulae of the form *(forall ??var ??phi)* and if *??phi* is a mental attitude of the agent. The variable *??var* is changed into Metavariable in the formula *??phi* and the method query is called on the resulting formula (*form*). If the result to the query is null and if *form* is a closed formula, the filter returns an empty list of `MatchResult`. Otherwise, it returns `null`.

IntentionTransferFilter

This filter is used to check if the semantic agent is cooperative towards another agent regarding a specific goal or not. This filter uses the `acceptIntentionTransfer` method (see paragraph 7) of the `StandardCustomization` class to check if the filter returns true or false. The filter is applicable on formulae of the form *(or (not (I ??agent1 ??goal)) (I ??agent2 ??goal))*, meaning *agent1* has the intention of *goal* implies *agent2* has the same intention. For example, this filter may be applied when the Jade agent receives a `Request`, a `Query-if`, a `Query-ref`, a `CallForProposal`, a `Confirm`, a `Disconfirm`, or an `Inform` message with the appropriate content.

IREFilter

This filter applies when the query relates to the equality between an identifying expression and a term. The filter is applicable on formulae of the form on the patterns *(B ??agent (= ??ire ??term))* and *(B ??agent (not (= ??ire ??term)))*.

OrFilter

This filter is applicable when the queried formula is an alternative (the filter is applicable on the formulae of the form *(or ??phi ??psi)*). The query method is called on phi and then on psi. The result is the union of the two resulting list of `MatchResults`.

Example of an applicative filter

Here is an example of Query filter:

```
((FilterKBase)myKBase).addKBQueryFilter(
new KBQueryFilterAdapter("(B ??agent (temperature_gt ??x))") {
  public QueryResult apply(Formula formula, Term agent) {
    QueryResult queryResult = new QueryResult();
    try {
      SLPatternManip.set(pattern, "agent", agent);
      Formula IOTA_VALUE = SLPatternManip.fromFormula((iota ?y (B ??agent
temperature ?y))));
     Formula IOTA_VALUE_GT = SLPatternManip.fromFormula((iota ?y (B ??agent
temperature_gt ?y))));
      // Test i the filter is applicable
      MatchResult applyResult = SLPatternManip.match(pattern, formula);
      if (applyResult != null && applyResult.getTerm("x") instanceof Constant) {
         Long queriedValue = ((Constant)applyResult.getTerm("x")).intValue();
         // query the base on temperature
         ListOfTerm queryRefResult = myKBase.queryRef((IdentifyingExpression)
             SLPatternManip.instantiate(IOTA_VALUE,"agent",applyResult.getTerm("agen
t")));
        if (queryRefResult != null && queryRefResult.size() != 0 ) {
          // The agent knows a value for the temperature predicate
          // Compares the known value and the queried value
    if ( ((Constant)queryRefResult.get(0)).intValue().longValue() >
queriedValue.longValue() ) {
                    // The queried value is greater than the known one
                    queryResult.setResult(new ListOfMatchResults());
            }
        } else {
            // query the base on temperature_gt
            queryRefResult = myKBase.queryRef((IdentifyingExpression)
                      SLPatternManip.instantiate(IOTA_VALUE_GT,"agent",
                                                 applyResult.getTerm("agent")));
             if (queryRefResult != null && queryRefResult.size() != 0 ) {
          // The agent knows a value for the temperature predicate
          // Compares the known value and the queried value
                 if (((Constant)queryRefResult.get(0)).intValue().longValue() >=
queriedValue.longValue() ) {
                    // The queried value is greater than the known one
                   queryResult.setResult(new ListOfMatchResults());
               }
             }
        }
        // The filter is applicable so the boolean value of the QueryResult is set
to true
        queryResult.setFilterApplied(true);
     }
   } catch (Exception e) {
      e.printStackTrace();
   }
   return queryResult;
  }
});
```

This filter is called when querying a fact matching the pattern `(B ??agent (temperature_gt ??x))`. The filter searches for a stored fact of the form `(temperature ??y)`; if it is found, and the `y` value is greater than x, the filter returns true. Else, it searches for a stored fact of the form `(temperature_gt ??y)`; if it is found, and the `y` value is greater than `x`, the filter returns true. Else the filter returns false.

5.2.1.3 FiltersDefinition class

An easy way to define filters is to use the `FiltersDefinition` class. This class is very useful to gather filters that deal with the management (i.e. assertion and querying) of specific predicates. The class extends `jade.leap.ArrayList.` An instance of such class is a sorted list of `FilterDefinition`, which is a filter associated at its index in the belief base list of filters. The `FiltersDefinition` class provides the following specific methods:

- The `defineFilter(int, KBFilter)` method adds a new filter in the list with a specified index;
- The `defineFilter(KBFilter)` method adds a new filter at the beginning of the list;
- The `get(int)` method returns the `FilterDefinition` instance at the specified index.

The `SingleNumValueDefinition` class of the temperature demo is a good example of use of this class. This class defines an unspecified predicate, which has only one numerical value as parameter.

```
public class SingleNumValueDefinition extends FiltersDefinition {

    //
    //Patterns used to manage the defined predicate
    //
    Formula VALUE_X_PATTERN;
    Formula NOT_VALUE_X_PATTERN;
    Formula VALUE_GT_X_PATTERN;
    Formula NOT_VALUE_GT_X_PATTERN;
    IdentifyingExpression ALL_VALUES;
    IdentifyingExpression ALL_VALUES_GT;
    IdentifyingExpression ALL_VALUES_NOT_GT;

    // If the parameter name takes the value "temperature", we find
    // the predicates handled in our example
    public SingleNumValueDefinition(String name) {
        VALUE_X_PATTERN = SLPatternManip.fromFormula("("+name+" ??X)");
        NOT_VALUE_X_PATTERN = SLPatternManip.fromFormula("(not ("+name+" ??X))");
        VALUE_GT_X_PATTERN = SLPatternManip.fromFormula("("+name+"_gt ??X)");
        NOT_VALUE_GT_X_PATTERN = SLPatternManip.fromFormula("(not ("+name+"_gt ??
X))");
     ALL_VALUES = (IdentifyingExpression)SLPatternManip.fromTerm("(all ?y
("+name+" ?y))");
    ALL_VALUES_GT = (IdentifyingExpression)SLPatternManip.fromTerm("(all ?y
("+name+"_gt ?y))");
     ALL_VALUES_NOT_GT = (IdentifyingExpression)SLPatternManip.fromTerm("(all ?y
(not ("+name+"_gt ?y)))");

      // ASSERT FILTERS
      // --------------
      // These filters are used to let only one information about this
      //predicate in the base.
      defineFilter(new KBAssertFilterAdapter("(B ??agent " + VALUE_X_PATTERN + ")") {
           ...
      });
      defineFilter(new KBAssertFilterAdapter("(B ??agent "+VALUE_GT_X_PATTERN+ ")") {
           ...
      });
      defineFilter(new KBAssertFilterAdapter("(B ??agent " + NOT_VALUE_GT_X_PATTERN
+ ")") {
      });

       // QUERY FILTERS
      defineFilter(new KBQueryFilterAdapter("(B ??agent " + VALUE_GT_X_PATTERN +
")") {
           ...
      });
      defineFilter(new KBQueryFilterAdapter("(B ??agent " + NOT_VALUE_GT_X_PATTERN +
")") {
           ...
        });
    }
}
```

5.2.1.4 *Adding new filters*

Adding new filters is done in the `setupKbase()` method of the class `SemanticCapabilities`.

```
public void setupKbase() {
      super.setupKbase();
     ((FilterKBase)myKBase).addKBAssertFilter(
              new KBAssertFilterAdapter("(B ??agent ??phi)") {
                      public Formula doApply(Formula formula) {
                      }
});
```

First, it is important to call the `super.setupKbase()` method to load all the generic filters used by the JSA. Then, the programmer can use the `addKBAssertFilter`, the `addKBQueryFilter` methods to add his/her own applicative filters.

5.2.1.5 *Removing beliefs from the base*

There is two ways to remove beliefs by asserting formulae:

- The first way is to assert a formula of the form `(not (B jsa ??phi))` where `??phi` is the formula to remove. In this case, the formula is simply removed from the base (if the formula is in the base). The programmer can use the `retractFormula(Formula)` method which applies this principle.
- The second way is to assert a formula of the form `(not ??phi)` where `??phi` is the formula to remove. As the base must remain consistent, the formula `??phi` is removed and the formula `(not ??phi)` is asserted.

5.2.2 Observers

The aim of observers is to observe the belief base and trigger some code if a specific formula (one formula per observer) value changes. Observers are defined by the interface Observer, which provides several methods:

- The `notify(ListOfMatchResults)` method notifies the observer that the value has changed;
- The `getObservedFormula()` method returns the observed formula.

The `ObserverAdapter` class is the implementation of the `Observer` interface provided by the framework. This class does nothing particular. At least, the `EventCreationObserver` class is used to create a special observer, which triggers the interpretation of a specified formula. A boolean given in the constructor of the class makes it possible to specify if the observer must trigger once or at each time the formula changes.

`Observers` are stored in the filter belief base in a list (`jade.leap.ArrayList`) by means of the `Observation` inner class. An `Observation` gathers an observer and the last observed value related to the observed formula. When a new value is asserted, the value is compared to the stored value in the corresponding `Observation` depending of the observed formula, and, if necessary, is updated. In case of change, the `notify` method is called.

Adding new observers is done in the redefinition of the `setupKbase` method of the `SemanticCapabilities` class.

```
public void setupKbase() {
    super.setupKbase();
...
    getMyKBase().addObserver(new EventCreationObserver(...));
...
}
```

Of course, the observers could be added dynamically by using the same method (for example, see the code of `semantics.interpreter.sips.Subscription` class code).

The conjunction of filters and observers makes it possible to implement the `Subscribe`, `Request-When`, and `Request-Whenever` act performances. For example, an agent *agt1* sends a `Request-When` to an agent *agt2* with the content *((action agt2 (INFORM :sender agt2 :receiver (set agt1) :content "((temperature 10))")) (temperature 10)))*. This message means that agent *agt2* should inform agent *agt1* that temperature is 10 when temperature is 10. The `Inform` act should be done only once. When *agt2* receives this message, a new `EventCreationObserver` *o* is added to its belief base (by means the `RequestWhen` SIP; see paragraph 9.2) to observe the formula *(temperature 10)*. If this formula is asserted in the *agt2* belief base, the `ObserverFilter` is tested. If the value has changed, the observer *o* is notified. As it is a `Request-When` act, the observer is removed from the belief base. A new event is generated on *(action agt2 (INFORM :sender agt2 :receiver (set agt1) :content "((temperature 10))")*. The effect of this will be the `Inform` message from *agt2* towards *agt1*.

By using correctly the observers, it is easy to specify behaviours related to changes on the agent's beliefs. For example, the behaviour consisting in taking off or putting on clothing in the temperature demo is due to the use of observers.

```
// Adds Observers to test if the temperature is greater than a specified level
//
// level 20
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))))))"
)));

// level 15
getMyKBase().addObserver(new EventCreationObserver(myAgent,
   SLPatternManip.fromFormula("(B "+getAgentName()+" (temperature_gt 15.0))"),
   SLPatternManip.fromFormula("
   (and (I "+getAgentName()+" (not (wearing "+getAgentName()+" pullover)))" +
   "(and (I "+getAgentName()+" (not (wearing "+getAgentName()+" coat)))" +
   "(I "+getAgentName()+" (not (wearing "+getAgentName()+" cap)))))")));

// level 10
getMyKBase().addObserver(new EventCreationObserver(myAgent,
   SLPatternManip.fromFormula("(B "+getAgentName()+" (temperature_gt 10))"),
   SLPatternManip.fromFormula("
   (and (I "+getAgentName()+" (not (wearing "+getAgentName()+" coat)))" +
   "(I "+getAgentName()+" (not (wearing "+getAgentName()+" cap))))))")));

// level 0
getMyKBase().addObserver(new EventCreationObserver(myAgent,
  SLPatternManip.fromFormula("(B "+getAgentName()+" (temperature_gt 0))"),
 SLPatternManip.fromFormula("(I "+getAgentName()+" (not (wearing "+getAgentName()+"
cap)))")));


// Adds Observers to test if the temperature is lower than a specified level
//

// level 20
getMyKBase().addObserver(new EventCreationObserver(myAgent,
 SLPatternManip.fromFormula("(B "+getAgentName()+" (not (temperature_gt 20)))"),
 SLPatternManip.fromFormula("(I "+getAgentName()+" (wearing "+getAgentName()+"
trousers))")));

// level 15
getMyKBase().addObserver(new EventCreationObserver(myAgent,
 SLPatternManip.fromFormula("(B "+getAgentName()+" (not (temperature_gt 15)))"),
 SLPatternManip.fromFormula("
   (and (I "+getAgentName()+" (wearing "+getAgentName()+" pullover))" +
   "(I "+getAgentName()+" (wearing "+getAgentName()+" trousers)))")));

// level 10
getMyKBase().addObserver(new EventCreationObserver(myAgent,
 SLPatternManip.fromFormula("(B "+getAgentName()+" (not (temperature_gt 10)))"),
 SLPatternManip.fromFormula("
   (and (I "+getAgentName()+" (wearing "+ getAgentName()+" coat))" +
   "(and (I "+getAgentName()+" (wearing "+getAgentName()+" pullover))" +
   "(I "+getAgentName()+" (wearing "+getAgentName()+" trousers))))"
)));

// level 0
getMyKBase().addObserver(new EventCreationObserver(myAgent,
 SLPatternManip.fromFormula("(B "+getAgentName()+" (not (temperature_gt 0)))"),
 SLPatternManip.fromFormula("
   (and (I "+getAgentName()+" (wearing "+getAgentName()+" cap))" +
   "(and (I "+getAgentName()+" (wearing "+ getAgentName()+" coat))" +
   "(and (I "+getAgentName()+" (wearing "+getAgentName()+" pullover))" +
   "(I "+getAgentName()+" (wearing "+getAgentName()+" trousers)))))")));
```

This code defines four temperature levels: 0, 10, 15, and 20. If the agent believes that it is above or below (one observer for each case) the one of these levels, it decides to put on or take off clothing. In details:

```
// Adds an Observer to the belief base
getMyKBase().addObserver(
    // It is an EventCreationObsever to trigger event
    new EventCreationObserver(
       //The agent that has this observer on its belief base
        myAgent,
       // The observed formula
        SLPatternManip.fromFormula("(B "+getAgentName()+ "(not (temperature_gt
20)))"),
      // The event to be generated
      SLPatternManip.fromFormula("(I "+getAgentName()+"(wearing " + getAgentName() +
" trousers))")));
```

This observer deals with the temperature lower than 20 degrees. The first formula means “the agent believes that the temperature is lower than 20 degrees”. The second one means “the agent has the intention to be dressed with his trousers”. The content of the intention corresponds to the rational effect of the `PUT-ON` ontological action (see paragraph 6.4) so that when the corresponding semantic event is triggered, the agent will perform the `PUT-ON` action in accordance with the rationality principle (see paragraph 9.2).

We focus now on the code of the `notify` method of the `EventCreationObserver`. The code is the following:

```
public void notify(ListOfMatchResults list) {
   try {
     if (list != null) {
       if (list.size() >= 1) {
          for (int i =0; i < ((MatchResult)list.get(0)).size(); i++) {
          // Instantiate the event to generate with the values of the match results
              SLPatternManip.instantiate(subscribedEvent,
              ((MetaTermReferenceNode)((MatchResult)list.get(0)).get(i)).lx_name(),
              ((MetaTermReferenceNode)((MatchResult)list.get(0)).get(i)).sm_value());
          }
       }
       //The agent interpretes the subscribe event
       myAgent.getSemanticCapabilities().getSemanticInterpreterBehaviour().interpret(
new SemanticRepresentation(subscribedEvent));
      // The case of Request When: the observer is deleted
      if (isOneShot)
myAgent.getSemanticCapabilities().getMyKBase().removeObserver(new Finder() {
            public boolean identify(Object object) {
                 return EventCreationObserver.this == object;
            }
         });
      }
   } catch (Exception e) {
        e.printStackTrace();
   }
}
```

This method uses the `interpret` method provided by the `SemanticInterpretationBehaviour` class, which is the mean to interpret correctly the given SR. This SR is added to the internal SR list of the behaviour and analysed using the SIPs as an external message.

The goal of an observer is to query the base after each assertion to test if the value of the observed formula has changed. Consequently lots of queries are done after each assertion. To optimize the performance, the programmer has to determine for each observer which patterns could have an effect on the value of the observed formula. By this way, when a new assertion is done, only the concerned observers are triggered. To reach this goal, the programmer has to override for each filter the `getObserverTriggerPatterns(Formula, Set)` method. The formula given in parameter is the one observed by the observer and the set given in parameter contains all the patterns which triggers the observer.

Let's take an example:

We consider the query filter which manages the `temperature_gt` predicate. Of course, the observer which observes the `(temperature_gt ??X)` pattern should be triggered at least if a formula matches this pattern. But, when you examine the code of the filter, you notice that the value of `temperature_gt` depends on the `(temperature ??X)` or the `(not (temperature_gt ??X))` patterns too. So, the observer on `(temperature_gt ??X)` should be triggers if there is modification on `(temperature ??X)` or the `(not (temperature_gt ??X))` patterns.

```
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();
   }
}
```

6 Semantic Actions

A Semantic Action is defined by:

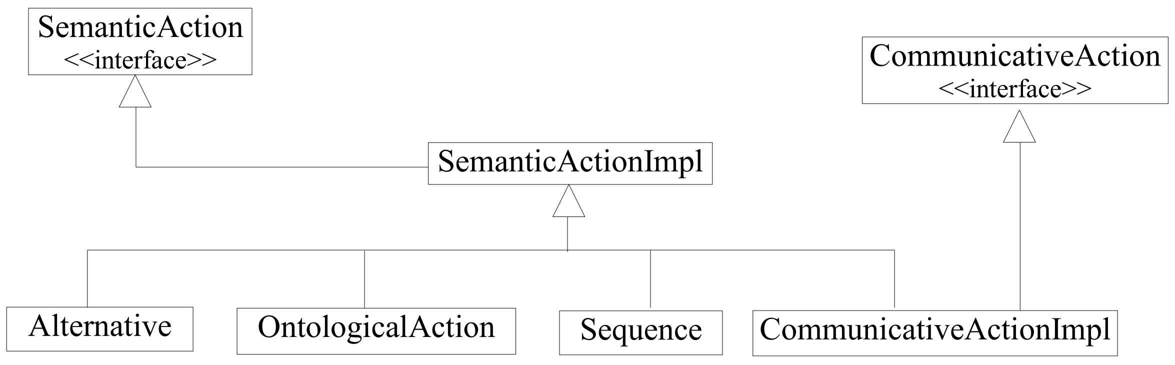
- a term that represents the author of the action;
- its feasibility precondition (represented by an SL formula), which represents a condition that must hold for an agent to be able to perform the action;
- its rational effect (represented by an SL formula), which represents a state intended by the agent performing the action;
- its persistent feasibility Precondition (represented by an SL formula), which represents the subset of the feasibility precondition that necessarily persists just after the performance of the action;
- its postcondition (represented by an SL formula), which represents the effect that the performing agent considers to be true just after the performance of the action;
- a semantic behaviour, which implements its performance by the agent.

The `SemanticAction` interface defines the methods to manage these characteristics:

- The `getAuthor()` method returns a Term that represents the author of the action;
- The `getFeasibilityPrecondition()` method returns the feasibility precondition;
- The `getRationalEffect()` method returns the rational effect of the action;

- The `getPersistentFeasibilityPrecondition()` method returns the persitentFeasibilityPrecondition;
- The `getPostCondition()` method returns the postcondition of the action;
- The `getBehaviour()` method returns the behaviour of the action;
- The `newAction(ActionExpression)` method creates a new instanced instance of the action based on the specified action expression;
- The `newAction(Formula, ACLMessage)` method creates a new instanced instance of the action based on the specified rational effect;
- The `toActionExpression()` method returns the action expression representation of this action.

The top of the hierarchy of semantic actions is shown in the next figure:



Three kinds of semantic actions are distinguished: the operators, the communicative actions, and the ontological actions. The alternative operator and the sequence operator are considered as "macro-actions", which express some complex actions by combining semantic actions. For example, the Inform-If act is formally defined as a finite alternative of two Inform acts. The communicative actions gather all the FIPA Communicative Acts. Finally, the ontological actions gather the actions related to a specific application.

6.1 Semantic action table

The semantic action table gathers all the semantic actions. The `SemanticActionTable` interface provides methods to handle the table:

- The `getSemanticCapabilities()` method returns the semantic capabilities that hold the action table;
- The `addSemanticAction(SemanticAction)` method adds a semantic action in the table;
- The `removeSemanticAction(Finder)` method removes the semantic action identified by the finder;
- The `getSemanticActionInstance(ActionExpression)` method creates an instantiated `SemanticAction` from the `SemanticAction` prototype within the table corresponding to an `ActionExpression`;
- The `getSemanticActionInstance(ACLMessage)` method creates an instantiated `SemanticAction` from the `SemanticAction` prototype within the table corresponding to an ACL Message;

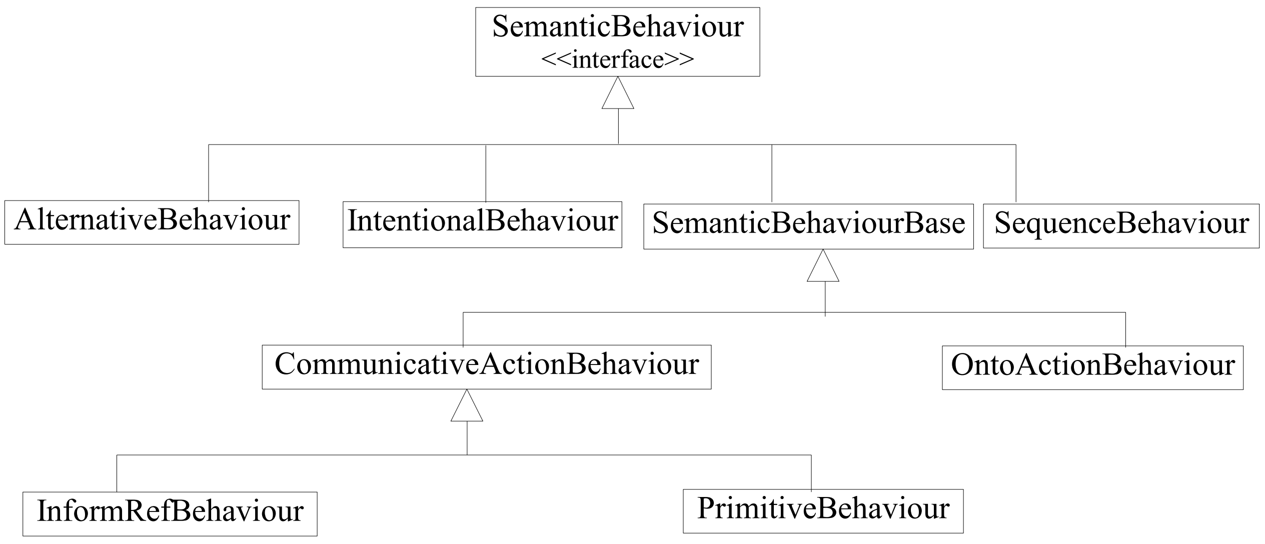
- The `getSemanticActionInstance(ArrayList, Formula, ACLMessage)` method creates a list of instantiated semantic actions from the `SemanticAction` prototypes within the table, such that each `SemanticAction` has a specified rational effect;
- The `loadTable()` method loads the actions in the table;
- The `size()` method returns the size of the table.

The position of an action in the table does not have any importance. The `SemanticActionTableImpl` class is the implementation of the `SemanticActionTable` interface provided by the framework. It extends the `jade.leap.ArrayList` class.

6.2 Semantic action behaviours

The definition of each semantic action must provide a semantic behaviour implementing the performance of this action. A semantic behaviour (see the `SemanticBehaviour` interface) is a classical JADE `Behaviour` with the additional property that they must end up in three possible ways:

- *success*, that means that the corresponding semantic action has been correctly performed;
- *feasibility failure*, that means that it is not possible to perform the action because its feasibility precondition is not satisfied;
- *execution failure*, that means that an unexpected failure is encountered during the performance of the action.



Of course, the hierarchy is close to the one of the actions. The behaviours attached to each semantic action implement the `SemanticBehaviour` interface. For each action is associated its behaviour:

- The `AlternativeBehaviour` for the `Alternative` operator;
- The `SequenceBehaviour` for the `Sequence` operator;
- The `CommunicativeActionBehaviour` for the `CommunicativeActionImpl` class;
- The `OntoActionBehaviour` for the `OntologicalAction` class.

AlternativeBehaviour

This behaviour extends `jade.core.behaviours.SequentialBehaviour`. It executes its sub-behaviours sequentially and terminates when:

- a behaviour of the sequence returns *success*. The internal state of the `AlternativeBehaviour` is set to *success*;
- a behaviour of the sequence returns *execution failure*. The internal state of the `AlternativeBehaviour` is set to *execution failure*;
- all behaviours of the sequence return *feasibility failure*. The internal state of the `AlternativeBehaviour` is set to *feasibility failure*.

SequenceBehaviour
This behaviour extends `jade.core.behaviours.SequentialBehaviour`. It executes its sub-behaviours sequentially and terminates when:

- a behaviour of the sequence returns *feasibility failure*. The internal state of the `SequenceBehaviour` is set to *success*;
- a behaviour of the sequence returns *execution failure*. The internal state of the `SequenceBehaviour` is set to *execution failure*;
- all behaviours of the sequence return *success*. The internal state of the `SequenceBehaviour` is set to *success*.

SemanticBehaviourBase
This abstract class models a basic implementation of the `SemanticBehaviour` interface. It extends `jade.core.behaviours.Behaviour`.

CommunicativeActionBehaviour
This abstract class models communicative behaviours. The `action` method follows these steps:

- if the current agent is the one making the action;
  - tests the `compute` method. This method should be overridden;
  - if the method returns true;
    - in this case, sends the ACL message corresponding to the action;
    - stores in the belief base the postconditions of the action;
    - stores in the belief base the belief of the agent on the fact that it has done the action;
    - sets the state of the behaviour to *success*;
  - if the method returns false;
    - sets the state of the behaviour to *feasibility failure*;
- on the contrary, does nothing.

If an exception occurs, sets the state to *execution failure*.

InformRefBehaviour
The Inform-Ref behaviour mainly consists in querying the belief base of the agent with the query expressed in the content. If the content is a *(all ?X ...)* expression, the query to the belief base necessarily returns a result o, which is in fact an SL object of type “`set`” (possibly empty). In this case, the performance of the `Inform-Ref` is always feasible.
If the content is a *(iota ?X ...)* or a *(any ?X ...)* expression, then the query to the belief base may either return a result *o* or return no result (meaning that the agent cannot answer the query, based on its current beliefs). In the first case, the performance of the `Inform-Ref` is feasible and consists in sending an `Inform` performative, which has an equals formula f (with the result *o*) in its content. In the second case, the performance of the `Inform-Ref` is not possible. If the simplification of the formula f match the pattern *(or (not (I ??agent1 (done ??act ??condition))) (I ??agent2 (done ??act*

??condition))) instead of sending an `Inform` message, a `propose` message is sent. This case has been introduced to work around a bug in the FIPA specification.

PrimitiveBehaviour

Basically, the performance of a primitive performative (`Inform`, `Request`, `Confirm` or `Disconfirm`) consists in sending the corresponding message. The compute method only checks the feasibility precondition of the action. It requires an access to the semantic action table to get this precondition, as well as an access to the belief base to check their satisfaction.
We point out that the majority of the actions are defined by means of the primitive performatives.

OntoActionBehaviour

This behaviour models the performance of any ontological action. If the action is feasible; the action is performed and the belief of feasibility precondition and belief of postcondition are considered as internal events. The belief of the agent on the fact that it has done the action is stored in the belief base.
This behaviour is automatically associated with an ontological action when this one extends the `OntologicalAction` class (see next paragraph).

IntentionalBehaviour

This behaviour his mainly used by the `ActionPerformance`, the `RationalityPrinciple`, and the `Planning` SIP to handle correctly the intentions of the agent. This behaviour extends `jade.core.behaviours.SequentialBehaviour`, which has only one sub-behaviour. The `onEnd` method is overridden like this:

- If the sub-behaviour finishes with success, this behaviour interprets the feedback if needed, and sets its internal state to *success*.
- If the behaviour finishes with *feasibility failure*, the same intention is reinserted to be interpreted by an other semantic interpretation principle with an upper index. In this case, the state of the behaviour is set to *feasibility failure*;
- Finally, if the sub-behaviour finishes with *execution failure*, the state is set to *execution failure*.

6.3 Communicative actions

This section reminds the reader of the list of communicative acts with a brief description. For each action corresponds a class, whose name is the one of the action.

- **Accept Proposal**: it is the action of accepting a previously submitted proposal to perform an action;
- **Agree**: the action of agreeing to perform some action, possibly in the future. This action is the general-purpose agreement to a previously submitted Request to perform some action. The agent sending the agreement informs the receiver that it does intend to perform the action, but not until the given precondition is true;
- **Cancel**: the action of one agent informing another agent that the first agent no longer has the intention that the second agent perform some action;
- **Call For Proposal (CFP)**: the action of calling for proposals to perform a given action. The content of this action contains an action expression denoting the action to be done, and a referential expression defining a single-parameter proposition which gives the preconditions of the action;

- **Confirm**: the sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition;
- **Disconfirm**: the sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true;
- **Failure**: the action of telling another agent that an action was attempted but the attempt failed. Failure is an abbreviation for informing that an act was considered feasible by the sender, but was not completed for some given reason. The first part of the content is the action not feasible. The second part is the reason for the failure, which is represented by a proposition. It may be the constant true;
- **Inform**: the sender informs the receiver that a given proposition is true. The content of this action is a proposition;
- **Inform If**: the sender informs the receiver whether or not a given proposition is true. The content of this action is a proposition;
- **Inform Ref**: the sender informs the receiver the object which corresponds to a descriptor, for example, a name. The content of this action is an object proposition (a referential description);
- **Not Understood**: the sender of the act (for example agt1) informs the receiver (for example agt2) that it perceived that agt2 performed some action, but that agt1 did not understand what agt2 just did. A particular common case is that agt1 tells agt2 that agt1 did not understand the message that agt2 has just sent to agt1. The first element of the message content is the action agt1 has not understood. The second element is a proposition representing the reason for the failure to understand.
- **Propose**: the action of submitting a proposal to perform a certain action, given certain preconditions. The content contains an action description, representing the action that the sender is proposing to perform, and a proposition representing the preconditions on the performance of the action;
- **Query If**: this action is the action of asking another agent whether or not a given proposition is true. The content of this action is a proposition;
- **Query Ref**: this action is the action of asking another agent for the object referred to by a referential expression. The content of this action is a descriptor (a referential expression).
- **Refuse**: the action of refusing to perform a given action, and explaining the reason for the refusal. The agent receiving a Refuse act is entitled to believe that:
    - the action has not been done,
    - the action is not feasible (from the point of view of the sender of the refusal), and,
    - the reason for the refusal is represented by a proposition which is the second element of the content (which may be the constant true).
- **Reject proposal**: the action of rejecting a proposal to perform some action during a negotiation. It is a general-purpose rejection to a previously submitted proposal. The agent sending the rejection informs the receiver that it has no intention that the recipient performs the given action under the given preconditions;
- **Request**: the sender requests the receiver to perform some action. The content of this action is an action expression;
- **Request When**: the sender wants the receiver to perform some action when some given proposition becomes true. The content contains a description of the action to perform, and the proposition;
- **Request Whenever**: the sender wants the receiver to perform some action as soon as some

proposition becomes true and thereafter each time the proposition becomes true again. The content of this action is a t-uple of an action expression and a proposition;
- **Subscribe**: the act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.

6.4 *Adding new actions*

The programmer has to override the `setupSemanticActions` method of the `SemanticCapabilities` class. The `addSemanticAction` of the `SemanticActionTable` interface is the method to use to add a new action:

```
public void setupSemanticActions() {
        super.setupSemanticActions();
        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) {
                        ...
           }
       });

           ...
}
```

It is highly recommended to call the method `super.setupSemanticActions`, which makes it possible to create a new instance of semantic actions table and to load the communicative actions. Without it, the created agent will not have any communicative action if the user does not load them explicitly and then the JSA is unable to automatically communicate with other agents using the framework mechanisms. So, if the programmer uses another implementation of Semantic Action table, the set-up of the table could be:

```
public void setupSemanticActions() {
          mySemanticActionTable = new mySemanticActionTableImpl(this);
          mySemanticActionTable.loadTable();
          //Ontological actions
          ...
}
```

Ontological actions

The framework defines all the actions defined by the FIPA-ACL Specifications (gathered in the `jade.semantics.actions.performatives` package). Of course, it is possible to define and add to the semantic action table, ontological actions relative to a specific application. The `OntologicalAction` class is there to help the programmer writing this kind of actions prototype. The constructor has four parameters:
- the action table which this action prototype belongs to;
- the pattern used to recognize the SL functional term representing this action;

- the pattern used to both recognize SL formulae representing the rational effect of this action and instantiate the SL formula representing the postcondition of this action;
- the pattern used to instantiate the SL formula representing the precondition of this action.

The next example shows a way to create such prototype:

```
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);
      }
}
```

All the metavariables of these patterns must refer to SL terms representing one of the arguments of the action and must use the same names for these metavariables. These patterns may refer to the reserved metareference "*??sender*", which denotes the agent of the action. A call to one of the `newAction` methods creates instances of this ontological action prototype.

The behaviour associated to this action is provided by the `perform` method:

```
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);
      }
}
```

In this example, the perform method, which holds the Java code of the action, simply calls a method of the agent. This method must be developed along the same way as the `action` method of the Jade Behaviour class. This method is called by the `OntoActionBehaviour` associated to the ontological action.

The `OntologicalAction` class provides several specific methods:

- The `perform(OntoActionBehaviour)` method is the implementation of the behaviour of the ontological action. This method must be developed along the same way as the `action` method of the Jade behaviour. This method must be overridden in all the subclasses (by default, does nothing but sets the internal state to *success* constant).
- The `getActionParameter(String)` method returns a `Term` representing a parameter from the given parameter name.

7 StandardCustomization

The `StandardCustomization` interface defines the methods to customize the JSA.

The first elements that can be customized this way are the Belief transfer and the Intention transfer principles. Two methods can be implemented to do so:

- `accceptBeliefTransfer(Formula, Term)`
- `acceptIntentionTransfer(Formula, Term)`

In both cases, the 2 arguments are:

- The `formula` representing the belief to be asserted or the intention to be adopted;
- The `agent` who holds the original belief or intention.

Based on this information, a semantic agent can decide to accept or reject beliefs and intentions of other agents. The next example illustrates how the belief transfer can be customized:

```
setMyStandardCustomization(new StandardCustomizationAdapter() {
  public boolean acceptBeliefTransfer(Formula formula, Term agent) {
      return  (SLPatternManip.match("(temperature ??x)", formula)==null)
          &&(SLPatternManip.match("(not (temperature ??x))", formula)==null)
          &&(SLPatternManip.match("(temperature_gt ??x)", formula)==null)
          &&(SLPatternManip.match("(not (temperature_gt ??x))", formula)==null);
      }});
```

The `acceptBeliefTransfer` method of the sensor agent is implemented so that the agent only believes facts from other agents that have nothing to do with the temperature. The second example illustrates how the intention transfer can be customized:

```
setMyStandardCustomization(new StandardCustomizationAdapter() {
      public boolean acceptIntentionTransfer(Formula goal, Term agent) {
          String motherID = "(agent-identifier :name "+motherAID.getName()+")";
          return agent.equals(SLPatternManip.fromTerm(motherID))
      }});
```

The `acceptIntentionTransfer` of the son agent is implemented so that the agent only adopts intention of his mother. Consequently, if another agent requests him to do any action, he will refuse.

The `StandardCustomization` object is also the means to properly handle some communicative acts like the `CFP` or the `Propose` acts. Due to the formal semantics of these acts, there is no means to automatically compute the JSA reaction. The provided methods are the following:

- The `handleCFPIota(Variable, Formula, ActionExpression, Term)` method returns a list of elements that corresponds to the answer to the query. One and only one solution is awaited;
- The `handleCFPAny(Variable, Formula, ActionExpression, Term)` method returns a list of elements that corresponds to the answer to the query. One solution is awaited;
- The `handleCFPAll(Variable, Formula, ActionExpression, Term)` method returns a list of elements that corresponds to the answer to the query. All the solutions are awaited.
- The `handleCFPSome(Variable, Formula, ActionExpression, Term)` method returns a list of elements that corresponds to the answer to the query. All solutions are awaited.

For these three methods, the first parameter is the variable used within the formula. The second formula represents the condition of the CFP. The third parameter is the requested action to be performed. The last parameter is the agent who will perform the action if accepted.

- The `handleRefuse(Term, ActionExpression, Formula)` method returns true if this method trap the specified formula when an agent (the first parameter) is no longer committed to do an action (the second parameter) under a condition (the third parameter);

- The `handleRejectProposal(Term, ActionExpression, Formula)` returns true if the specified formula is trapped when an agent (the first parameter) is no longer interested to do an action (the second parameter) under a condition (the last parameter);
- The `handleAgree(Term, ActionExpression, Formula)` method returns true if the specified formula is trapped when an agent (the first parameter) is committed to do an action (the second parameter) under a condition (the last parameter);
- The `handleProposal(Term, ActionExpression, Formula)` method returns true if the specified formula is trapped when an agent (the first parameter) is making a proposal (of doing an action (the action the sender will perform in case of acceptance) (the second parameter) under a condition (the third parameter) (the formula that represents the proposal of the sender in order to perform the action) towards the Jade agent.

In the next example, the display agent handles the sensor's proposals in response to a previous CFP (in the `DisplayCapabilities` class of the demo package).

```
setMyStandardCustomization(new StandardCustomizationAdapter() {

public boolean handleProposal(Term agent, ActionExpression action, Formula formula)
{
        Term act = SLPatternManip.fromTerm("(action ??receiver (INFORM-REF
                            :content \"((any ?x (temperature ?x)))\" :receiver
(set ??agent)               :sender ??receiver))");
        MatchResult matchResult = SLPatternManip.match(act, action);
        if ( matchResult != null ) {
                Formula cond = SLPatternManip.fromFormula("(precision ??x)");
                matchResult = SLPatternManip.match(cond, formula);
                if ( matchResult != null ) {
                        ((DisplayAgent)myAgent) .handleProposal(
                        (IntegerConstantNode)matchResult.getTerm("x"),
                                agent, action, formula);
                }
        }
        return true;}}
);
```

The code of this method only checks if the action and the condition are the ones of the sent `CFP`. In this case, another method of the display agent is called that actually handle the proposal.

In the next example, the sensor agent handles the display's CFP message. The CFP condition is the formula `(precision ??X)` and the precision of the sensor is returned as solution.

```
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;
        }
}
```

Finally, this class provides two methods to notify an agent when a `Subscribe`, `RequestWhen`, `RequestWhenever`, or an `Unsubscribe` message occurs. That makes it possible to the programmer to insert Java code to realize specific actions related to the application.

- The `notifySubscribe(Term subscriber, Formula observed, Formula goal)` method notifies the agent that it has just receive a subscribe from the subscriber on the formula "observed" with the goal "goal";
- The `notifyUnsubscribe(Term subscriber, Formula observed, Formula goal)` method notifies the agent that it has just receive an unsubscribe from the subscriber on the formula "observed" with the goal "goal".

For example:

```
// Sets the colour to yellow when it receives a subscribe
public void notifySubscribe(Term subscriber, Formula obsverved, Formula goal) {
          ((SensorAgent)myAgent).setSubscribed(true);
}
//Sets the colour to gray when it receives an unsubscribe
public void notifyUnsubscribe(Term subscriber, Formula obsverved, Formula goal)
{
          ((SensorAgent)myAgent).setSubscribed(false);
}


void setSubscribed(boolean subscribed) {
          if ( subscribed ) {
                    mainPanel.setBackground(Color.YELLOW);
          }
          else {
                    mainPanel.setBackground(Color.GRAY);
          }
}
```

The `StandardCustomizationAdapter` provides a basic implementation of the `StandardCustomization` interface. By default, each method does nothing.

8 Planner

Planners are used to find a plan that reaches a given goal. No specific planner is provided by the framework. The programmer has to provide his/her own planner. This one must implement the interface `Planner`, which defines only one method: `findPlan(Formula, SemanticRepresentation)`. The method returns a Jade Behaviour corresponding to the execution of a plan which makes it possible to reach the given goal.

9 Semantic interpretation principles

The Semantic interpretation principles (SIPs) are related to the behaviour of the agent and provide the basic means to produce and consume SRs. Each SIP is applicable on a specific pattern (that should match the SL Formula of the incoming SR) and the application of each SIP results in querying or updating the agent's belief base and adding to, or removing behaviours from the agent.
A SIP is defined by a pattern (to test if it is applicable), the position of it in the table and the `SemanticCapabilities` object to which it belongs. The SIPS are stored in an **ordered** table.

9.1 Semantic interpretation principles table

The `SemanticInterpretationPrincipleTable` class defines the SIPs ordered list. This interface provides all the methods to manage the SIPs:

- The `addSemanticInterpretationPrinciple(SemanticInterpretationPrinciple)` method adds a new semantic interpretation principle at the end of the table. Sets the inner sip index at the correct value;
- The `addSemanticInterpretationPrinciple(SemanticInterpretationPrinciple , int)` method adds a semantic interpretation principle in the table at the specified index. Sets the inner sip index ;
- The `removeSemanticInterpretationPrinciple(Finder)` method removes all the semantic interpretation principles that correspond to the finder;
- The `getSemanticInterpretationPrinciple(int)` method returns the semantic interpretation principle at the specified index in the table;
- The `loadTable(SemanticCapabilities)` method loads all the semantic interpretation principles in the table for the specified agent;
- The `size()` method returns the size of the table;
- The `removeSemanticInterpretationPrinciple(int)` method removes the semantic interpretation principle at the specified index.

9.2 List of generic SIPs

In this section standard SIP provided within the framework are presented in the order they are in the table.

- **And**: This SIP is intended to be applied to an AND formula of the form *(and φ ψ)*. It produces two Semantic Representations: the left part of the conjunction (i.e. *φ*), and the right part of the conjunction (i.e. *ψ*).
- **EqualsIRE**: This SIP is applicable on formulae of the form *(B agent (= ??ire ??phi))*. The identifying expression *??ire* is of the form *(op t f)* where *op* is an operator of the set {`iota`, `all`}, *t* a term, and *f* a formula. *??phi* is a formula.

  If the operator is equal to `all`, and *??phi* is a no empty set, the sip returns new several SR:

  - a SR of the form *(B agent (= ??ire (set)))* that indicates the agent knows all values that make the formula f `true`.
  - for each value of the set, a SR of the form *(B agent f)* is generated, where the variables in *f* are replaced by the values of the set.

  If the operator is equals to `iota`, the sip returns two SR:

  - a SR of the form *(B agent (= (all t f) (set)))* that indicates the agent knows all values that make the formula f `true`.
  - a SR of the form *(B agent f)* is generated, where the variables in *f* are replaced by the corresponding values in ??*phi*.
- **ActionFeatures**: This principle is intended to be applied to the initial formula representing the fact that the Jade agent has perceived an incoming ACL message *(B ja (done (action s (PERFORMATIVE :sender s :receiver r :content "c")) true))*. It produces four new Semantic Representations :
  - one for checking the consistency. This SR is not expected to be used further by others

SIPs;

- one stating that the Jade agent believes the persistent feasibility precondition of the received message is satisfied.
- one stating the Jade agent believes the intentional effect of the received message.
- the last is the postcondition of an action, the performance of which has just been observed by the Jade agent.

- **AlreadyReachedGoal**: This SIP traps the intentions of the semantic agent that the semantic agent already believes. This thus makes it possible to avoid useless calculations. This SIP is applicable on formulae of the form *(I ja φ)*;
- **BeliefTransfer**: This semantic interpretation principle expresses a necessary cooperation principle of the Jade agent towards the beliefs that the sender (of an ACL message) intends to communicate. Typically, this SIP is used to interpret incoming Inform messages. Pragmatically, it infers a belief such as *(B ja φ)* from a previously deduced SL formula such as *(B ja (I s (B ja φ)))*. This inference is not always desirable, so that this step is actually applied under some conditions that evaluate if the Jade agent accepts a belief φ from an agent s. For example, the Jade agent may not trust certain agents and so may not come to believe necessarily what they tell. In order to avoid some new heterogeneous mechanism, this condition is evaluated by asking the belief base of the Jade agent if a certain formula holds, namely *(implies (I s (B ja φ)) (B ja φ))* , that is, does the Jade agent come to believe φ if an agent s intends him/her to do so;
- **RequestWhen**: This SIP enables the Jade agent to add an observer to the belief base. This observer can be triggered only once. This SIP is applicable on formula of the form *(B ?? agent (I subscriber (done act (and (B agent subscribeProperty) (exists ?e (done ?e (not (B agent subscribeProperty)))))))))*. The `apply` method calls the `notifySubscribe` method of the `StandardCustomization` class. The `RequestWhen` class inherits the `Subscription` class;
- **IntentionTransfer**: This SIP expresses a necessary cooperation principle of the Jade agent (receiving the ACL message) towards the intentions that the sender intends to communicate. This SIP is typically used to interpret incoming Request messages or Inform messages, the content of which being an intention of the sender. This SIP is actually applied under some conditions, which are evaluated by asking the belief base of the Jade agent with a question, regarding an intention transfer. When applied, this step actually infers two new SL formulae:
  - If the intention transfer is allowed, infers *(I agent φ)*, expressing the adoption of the intention by the Jade agent, and, optionally (depending on the targeted behaviour of the Jade agent), *(I agent (B s (I agent φ)))*, expressing the intention of the Jade agent to give a feedback of this intention transfer to the sender of the message (which should generally give rise to an Agree message towards the sender).
  - If the intention transfer is forbidden, infers *(not (I agent φ))* and, optionally, *(I agent (B s (not (I agent φ))))* (which should generally give rise to a Refuse message towards the sender);
- **Planning**: This principle provides the Jade agent with a more general mean of planning than the previous ones (rationality principle and action performance). It calls an external component that returns a Jade behaviour that implements a way to reach an input goal *φ*, and

adds this behaviour to the agent. As above, this behaviour is also encapsulated in an IntentionalBehaviour that memorizes the triggering intention and deletes it from the agent belief base if it succeeds. If no Jade behaviour can be found by the planning component, this step does nothing. This SIP is applicable on formulae of the form *(I agent φ)*;

- **ActionPerformance**: this SIP provides some basic planning for goals of the form *(done a true)*, where *a* is a (possibly complex expression of) action, the author of which is the Jade agent. It consists in adding to the Jade agent a Jade Semantic Behaviour performing the targeted semantic action *a*. This Behaviours is encapsulated into an IntentionalBehaviour (see paragraph 6.2) which memorizes the triggering intention (namely the goal *(done a true)*) so that it can delete it from the agent belief base if it succeeds in performing the action;
- **RationalityPrinciple**: This step enables the Jade agent to compute some trivial planning according to the rationality principle. This SIP is applicable on formulae of the form *(I agent φ)*. It looks for all the semantic actions available to the Jade agent, the rational effect of which matches the intention φ of the Jade agent. It then builds (and add to the agent) a Jade semantic behaviour implementing one of these actions (which is represented by an alternative action expression). This behaviour is encapsulates in an IntentionalBehaviour (see paragraph 6.2), which memorizes the goal φ, so that it can delete the corresponding intention from the agent belief base if it succeeds in reaching this goal. Obviously, this kind of planning can only find plans for goals that exactly match the rational effect of a communicative act known by the Jade agent. For example, from an intention of the form *(I agent (B r (p)))*, this step adds a Jade Behaviour implementing the alternative:

```
(| (action ja (INFORM :sender ja :receiver r :content "((p))"))
          (| (action ja (CONFIRM :sender ja :receiver r :content "((p))"))
          (action ja (DISCONFIRM :sender ja :receiver r :content "((not
(p)))"))
          )
)
```

  Indeed, the mentioned Inform, Confirm and Disconfirm acts are exactly the ones that the Jade agent knows (from his FIPA performative library) and that have *(B r (p))* as rational effect;

- **Refuse**: This principle is intended to be applied if an agent is no longer committed to do an action under a condition. This principle may be applied when the Jade agent receives a Cancel or a Refuse message. This SIP is applicable on formulae of the form *(not (I agent (done (action agent act) φ)))*. The apply method calls the handleRefuse method of the StandardCustomization class;
- **RejectProposal**: This principle is intended to be applied if an agent is no longer interested in the Jade agent doing an action under a condition. The Jade agent may then drop his behaviour that currently seeks performing this action. This principle may be applied when the Jade agent receives a Cancel or a RejectProposal message. This SIP is applicable on formulae of the form *(not (I agent1 (done (action agent2 act) φ)))*. The apply method of the SIP calls the method handleRejectProposal of the StandardCustomization class;
- **Agree**: This principle is intended to be applied if an agent is committed to do an action

under a condition. This principle may be applied when the Jade agent receives an `Agree` message. This SIP is applicable on formulae of the form *(I agent (done (action agent act) φ))*. The `apply` method of the SIP calls the method `handleAgree` of the `StandardCustomization` class;

- **Propose**: This principle is intended to be applied when an agent receives a Propose message. This SIP is applicable on formulae of the form *(B agent1 (or (not (I agent1 (done act φ))) (I agent2 (done act φ))))*. The `apply` method of the SIP calls the method `handleProposal` of the `StandardCustomization` class;
- **RequestWhenever**: This SIP enables the Jade agent to add an observer to the belief base. This SIP is applicable on formulae of the form *(or (not (B agent subscribeProperty)) (or (I subscriber goal) (forall ?e (not (done ?e (not (B agent subscribeProperty)))))))*. The `apply` method calls the `notifySubscribe` method of the `StandardCustomization` class. The `RequestWhenever` class inherits the `Subscription` class;
- **Subscribe**: This SIP enables the Jade agent to add an observer to the belief base. This SIP is applicable on formulae of the form *(or (I subscriber goal) (or (forall ?y (not (B agent ire))) (forall ?e (not (done ?e (forall ?y (not (B agent ire)))))))))*. The `apply` method calls the `notifySubscribe` method of the `StandardCustomization` class. The `Subscribe` class inherits the `Subscription` class;
- **Unsubscribe**: This principle is intended to be applied when an agent receives an `UnSubscribe` message. This SIP is applicable on formulae of the form *(B agent (or (not (B agent property)) (or (not (I subscriber goal)) (forall ?e (not (done ?e (not (B agent property))))))))* or of the form *(B agent (not (I subscriber goal)))*. If the SIP is applicable, it removes the observer related to the specified property. The `apply` method calls the `notifyUnsubscribe` of the `StandardCusomization` class;
- **UnreachableGoal**: This principle is intended to be applied to all intentions that have not been realised. These intentions are considered as not feasible. If the intention becomes from a request of another agent, a new SR is generated in order to inform the other agent, that the intention is not feasible. The new SL formula is of the form *(I agent (B r (forall ?e (not (B agent (feasible ?e φ))))))*, meaning the agent *agent* has the intention that *r* believes that *agent* thinks that there is not an unspecified event after which *φ* would be true. This SIP is applicable on formulae of the form *(I agent φ)*.

9.3 Adding new SIPs

The programmer has to override the `setupSemanticInterpretationPrinciples` method of the `SemanticCapabilities` class:

```
public void setupSemanticInterpretationPrinciples(){
    super.setupSemanticInterpretationPrinciples();
    getMySemanticInterpretationTable().addSemanticInterpretationPrinciple(
new mySIP(this));

}
```

It is highly recommended to call the method `super.setupSemanticInterpretationPrinciples()`, which makes it possible to create a new instance of SIP table and to load the generic SIPs. Without it, the created agent will not have any SIP if the user does not load them explicitly and then the JSA is unable to automatically communicate with other agents using the framework mechanisms.

The `SemanticInterpretationPrincipleTable` interface provides two constants to help the addition of new SIP:

- `Front` is used to add a SIP at the beginning of a list;
- `End` is used to add a SIP at the end of the list

Warning:
The programmer must be very careful if s/he wishes to add a new SIP. As the table of SIP is ordered, the place where the SIP is inserted must be chosen with caution. For example, the added SIP can catch formula and then stop the classical interpretation mechanism in some case it should not. A SIP that has a formula more specific that another must be placed before in the list.

In all the cases, it is necessary to use the `addSemanticInterpretationPrinciple` method (of the `SemanticInterpretationTable` class) in order to be sure that the inner index of each SIP is correct.

We advise programmers to add their applicative SIP at the beginning of the table.

9.4 Examples of semantic interpretations

This section presents examples of how the interpretation process based on the SIPs applies on FIPA acts with various contents. The next figure shows the interpretation process for a simple `Inform` message.

An incoming message is represented by the formula:

(B receiver (done (action sender message) true))

This formula states that the agent *receiver* believes the action of the agent *sender* sending the message *message* has just occurred. In the case presented in the example of figure 4, the agent *ja* believes that the agent *sender* has just informed it that the temperature is ten.

The SIP index of SR of an incoming message is always 0. That means that the SIPs are tested from the beginning of the table. The first SIP tested by the `SemanticInterpreterBehaviour` is `ActionFeature` SIP. It is applicable and produced four new SRs with their SL formula represented in the figure and with a SIP index set at 0. The feasibility precondition formula is not represented because it is never used further and its writing would have been too large for the figure. No SIP is applicable on the persistent precondition and the postcondition. These formulae are then asserted in the belief base. On the other hand, the `BeliefTransfer` SIP is applicable on the rational effect formula and produces a new SR (we suppose here that the belief transfer is possible between the two agents). No SIP is applicable on this new SR, which is asserted in the belief base. At least, the agent *ja* knows that the temperature is 10 as it has *(temperature 10)* in its belief base.

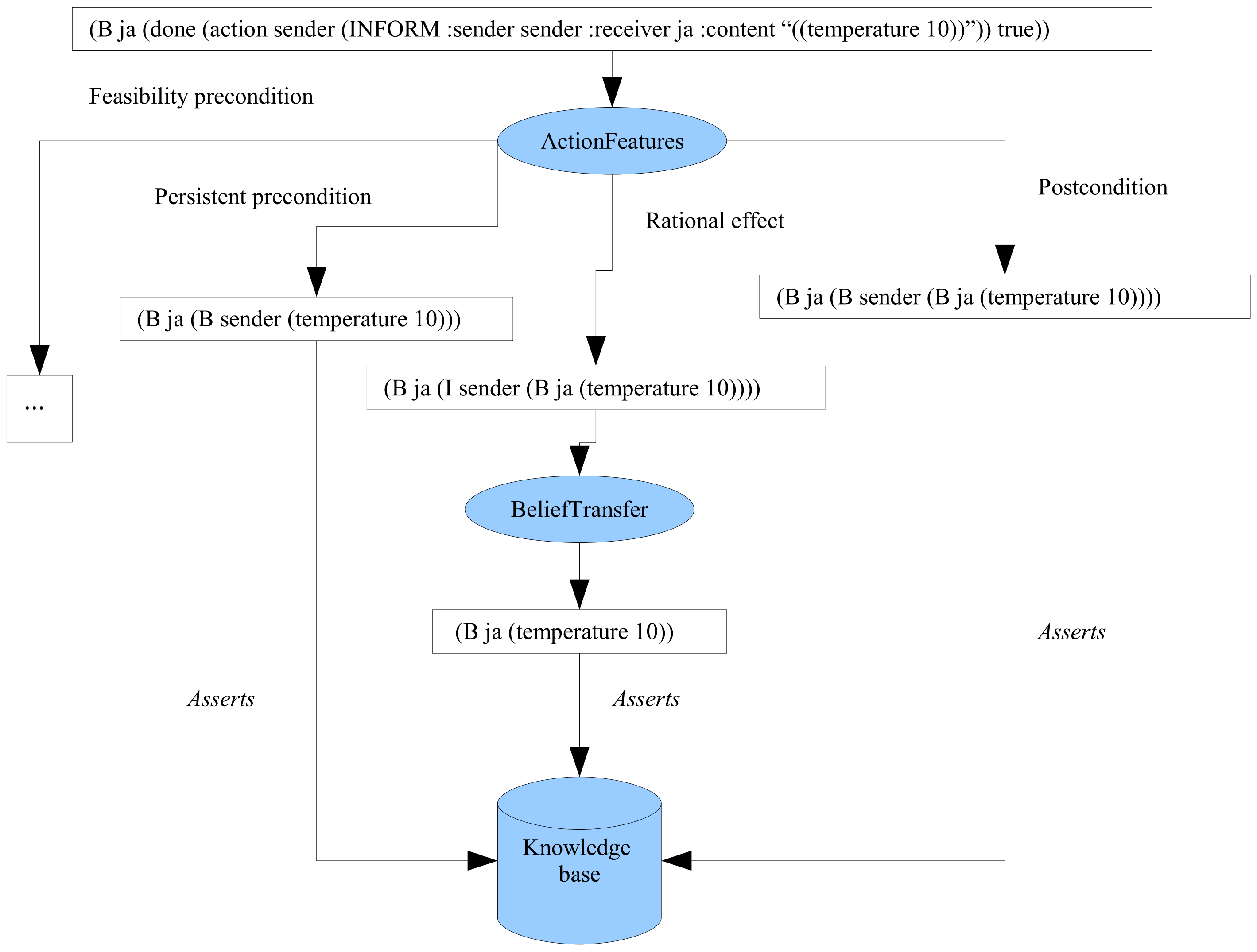

(B ja (done (action sender (INFORM :sender sender :receiver ja :content "((temperature 10))")) true))
Feasibility precondition
ActionFeatures
Persistent precondition
Rational effect
Postcondition
(B ja (B sender (temperature 10)))
(B ja (B sender (B ja (temperature 10))))
...
(B ja (I sender (B ja (temperature 10))))
BeliefTransfer
(B ja (temperature 10))
Asserts
Asserts
Asserts
Knowledge base

Figure 4: Example of an Inform message interpretation

Figure 5 shows the interpretation process for a `Request` message. For this purpose, let us consider the very simple ontological action `WAIT` (extracted from the temperature demo):

```
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: {
        // Adjust wakeupTime in case the user set a relative time
         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);
         behaviour.setState(1000);
         break;
        }
        case 1000: {
        // in this state the behaviour can be restarted for two reasons
        // 1. the timeout is elapsed and then 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 : {
            // this case should not occur
            behaviour.setState(SemanticBehaviour.EXECUTION_FAILURE);
            break;
        }
   } // end of switch
}
```

The postcondition and the precondition of this action are the formula “true”, meaning that there is not any special condition to do it. The `perform` method checks if the behaviour has just started or if it has already been block for the time period. In the first case, the perform method blocks the behaviour for the time delay. In the second case, it simply returns a successful state.

The following figure shows the interpretation process for a Request incoming message. The agent sender requests the agent *ja* to wait a while. The figure shows the typical case where the author of the requested action is the Jade agent. The action performance deductive step may then be applied so that the Jade agent adds a new semantic behaviour implementing the performance of the action. The figure also shows the usage of the rationality principle to trigger the performance of an Agree message.

If the author of the requested action is different from the Jade agent, then the rationality principle or the more general planning deductive step should be applied.

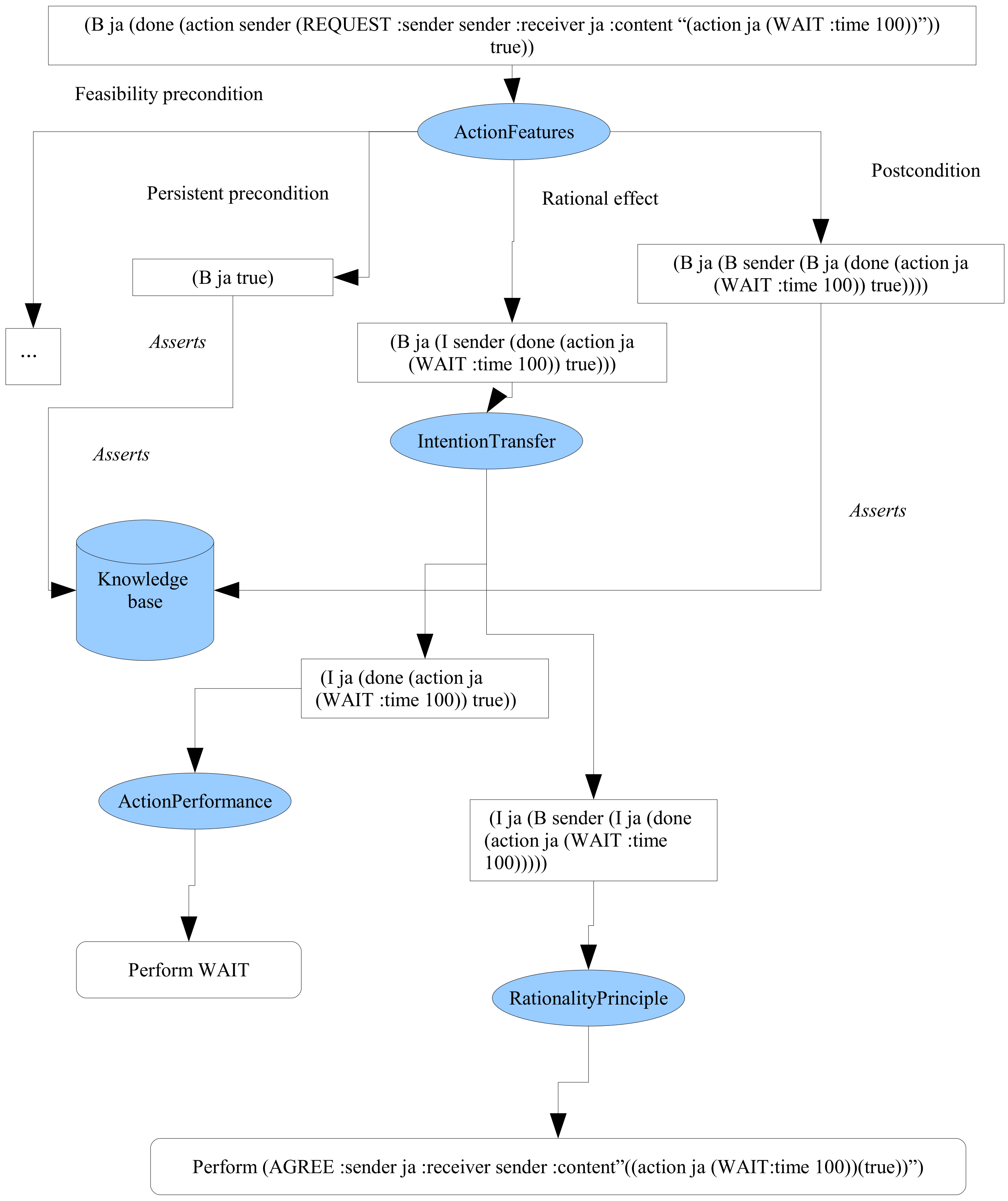

(B ja (done (action sender (REQUEST :sender sender :receiver ja :content "(action ja (WAIT :time 100))")) true))
Feasibility precondition
ActionFeatures
Persistent precondition
Rational effect
Postcondition
(B ja true)
(B ja (B sender (B ja (done (action ja (WAIT :time 100)) true))))
...
Asserts
(B ja (I sender (done (action ja (WAIT :time 100)) true)))
IntentionTransfer
Asserts
Asserts
Knowledge base
(I ja (done (action ja (WAIT :time 100)) true))
ActionPerformance
(I ja (B sender (I ja (done (action ja (WAIT :time 100)))))
Perform WAIT
RationalityPrinciple
Perform (AGREE :sender ja :receiver sender :content"((action ja (WAIT:time 100))(true))")

Figure 5: Example of a Request message interpretation

9.5 Example of applicative SIPs

Let us suppose that in an application, an agent needs to show a photo in a panel. It is a behavioural aspect of the agent and so should be done by a SIP. For example, the code could be the following:

```
public class ViewerSIPContent extends SemanticInterpretationPrinciple {
    private Formula pattern;
    private PhotoPanel photoPanel;

    public ViewerSIPContent(SemanticCapabilities capabilities, PhotoPanel pp) {
        super(capabilities);
        pattern = SLPatternManip.fromFormula("(B ??agent (image-content ??id ??
content))");
        photoPanel = pp;
    }

    public ArrayList apply(SemanticRepresentation sr)
            throws SemanticInterpretationPrincipleException {
        MatchResult applyResult = SLPatternManip.match(pattern,
sr.getSLRepresentation());
        if (applyResult != null) {
           try {
            byte[] bytes =
((ByteConstantNode)applyResult.getTerm("content")).lx_value();
            photoPanel.setPhoto(JPEGUtilities.load(new
ByteArrayInputStream(bytes)));
            return new ArrayList();
          } catch (SLPatternManip.WrongTypeException wte) {}
        }
        return null;
    }

}
```

In the constructor, the pattern which makes it possible to test the applicability of the SIP is set like the panel where the image should be shown.

In the apply method, the first step is to test if the SIP is applicable. If not, the method returns null.

If the SIP is applicable, it gets the content of the photo (the bytes array representing the photo) in the incoming formula and then sets the image in the panel. The method returns an empty list to indicate that the SIP applied. The list is empty because it is not wished that other SIP use this Semantic representation or that information is asserted in the base.

Let us suppose now that when a new photo is added, the name of the photo should appears in a list of the IHM. It is a another behavioural aspect of the agent and so should be done by a SIP. For example, the code could be the following:

```
public class ViewerSIPImgDescr extends SemanticInterpretationPrinciple {
    private Formula pattern;
    private PhotoPanel photoPanel;
    public ViewerSIPAllImgDescr(SemanticCapabilities capabilities, PhotoPanel pp) {
        super(capabilities);
        pattern = SLPatternManip.fromFormula("(B ??agent (= (all (sequence ??id ??
desc) (image-description ??id ??desc)) (set)))");
        photoPanel = pp;
    }

    public ArrayList apply(SemanticRepresentation sr)
            throws SemanticInterpretationPrincipleException {
        MatchResult applyResult = SLPatternManip.match(pattern,
sr.getSLRepresentation());
        if (applyResult != null) {
           try {
             photoPanel.addPhotoDescription(applyResult.get);
             ArrayList list = new ArrayList();
             sr.setSemanticInterpretationPrincipleIndex(sr.getSemanticInterpretation
PrincipleIndex() + 1);
             list.add(sr);
             return list;
           } catch (SLPatternManip.WrongTypeException wte) {}
        }
        return null;
    }
}
```

In this case, the constructor in the same as the previous one for another pattern.
If the SIP is applicable, the list of photos is modified. The method returns a SR list to indicate that the SIP applied. On the other hand, the same SR is added in this list because it is wished that the incoming formula be asserted in the belief base (and perhaps analysed by other SIP). However, its internal index is increased by 1 to prevent that this same SIP applies again.

10 Useful classes

10.1 Finder class

This class of the `jade.semantics.intrepeter` package represents a general object that permits object identification. It should be extended to handle the specific need of a particular identification. This class provides two methods:

- The `identify(Object)` method returns true if the object passed in parameter is identified by this identifier, false in the other case. By default, this method returns false. It should be overridden. For example,

```
((SemanticAgent)myAgent).getSemanticCapabilities().getMyKBase().removeFormula(
   new Finder() {
      public boolean identify(Object object) {
         Formula pattern = SLPattern.fromFormula("(B ??agent ??phi)");
         if (object instanceof Formula) {
           return (SLPatternManip.match(observedPattern, ((Formula)object)) !=
null);
           }
      }
);
```

  In this example, the finder looks for Formula that matches the pattern `(B ??agent ??phi)`. If it finds one of them, this one is then removed from the belief base.

- The `removeFromList(ArrayList)` method removes an object of the list if it is identified.

10.2 Tools class

The `Tools` class of the `jade.semantics.intrepeter` package provides useful methods:

- The `term2AID(Term)` method returns the AID corresponding to the term representing an agent;
- The `AID2Term(AID)` method returns the term representing an agent to the corresponding AID;
- The `isCommunicativeActionFromMeToReceiver(ActionExpression, Term, SemanticAgent)` method tests if the action expression given in parameter is a communicative action from the current semantic agent to the specified receiver.

10.3 Util class

The `Util` class of the `jade.semantics.lang.sl.tools` package provides useful method to handle SL formulas.

- The `buildAndNode(ListOfNodes)` method returns a `AndNode` formula built with the formulae of the given list. If the size of the list equals 1, it returns the only formula of the list (which is not necessary an `AndNode` formula). If the size of the list equals 0 or if the list is `null`, returns `null`.
- The `buildOrNode(ListOfNodes)` method returns a `OrFormula` formula built with the formulae of the given list. If the size of the list equals 1, it returns the only formula of the list. If the size of the list equals 0 or if the list is `null`, returns `null`.
- The `instantiateInMatchResult(MatchResult, String, Node)` method returns `true` if a metavariable of the given `MatchResult` has the given `varName` in and if it succeeds in giving it the given value.

11 Appendix

This paragraph presents the SL grammar.

```
----------------------------------------------------
--                    CONTENT
----------------------------------------------------
CONTENT ::= content
         => as_expressions : CONTENT_EXPRESSION_S
{
    String toSLString();
    Node getContentElement(int i);
    void setContentElement(int i, Node element);
    void addContentElement(Node element);
    void setContentElements(int number);
    int contentElementNumber();
    jade.semantics.lang.sl.tools.MatchResult match(Node expression);
    Node instantiate(String varname, Node expression);
};

content =>;
----------------------------------------------------
--                CONTENT_EXPRESSION
----------------------------------------------------
CONTENT_EXPRESSION_S ::= Seq of CONTENT_EXPRESSION;

CONTENT_EXPRESSION ::= action_content_expression
```

```
                       | identifying_content_expression
                       | formula_content_expression
                       | meta_content_expression_reference
{
   Node getElement();
};

action_content_expression => as_action_expression : ACTION_EXPRESSION;

identifying_content_expression       =>       as_identifying_expression       :
IDENTIFYING_EXPRESSION;

formula_content_expression => as_formula : FORMULA;

meta_content_expression_reference => lx_name : (java.lang.String),
                                      sm_value : CONTENT_EXPRESSION;

----------------------------------------------------
--                  FORMULA
----------------------------------------------------
FORMULA_S ::= Seq of FORMULA;

FORMULA ::= ATOMIC_FORMULA
            | UNARY_LOGICAL_FORMULA
            | MODAL_LOGIC_FORMULA
          | ACTION_FORMULA
            | QUANTIFIED_FORMULA
          | BINARY_LOGICAL_FORMULA
            | meta_formula_reference
         => sm_simplified_formula : FORMULA
{
    FORMULA getSimplifiedFormula();
    void simplify();
    boolean isMentalAttitude(TERM term);
    boolean isSubsumedBy(FORMULA formula);
    boolean isConsistentWith(FORMULA formula);
    FORMULA getDoubleMirror(TERM i, TERM j, boolean default_result_is_true);
    boolean isAFreeVariable(VARIABLE x);
    FORMULA getVariablesSubstitution(VARIABLE_S vars);
    FORMULA getVariablesSubstitutionAsIn(FORMULA formula);
    FORMULA getVariablesSubstitution(VARIABLE x, VARIABLE y);
    FORMULA isBeliefFrom(TERM agent);
    jade.semantics.lang.sl.tools.MatchResult match(Node expression);
    Node instantiate(String varname, Node expression);
};

meta_formula_reference => lx_name : (java.lang.String),
                           sm_value : FORMULA;

----------------------------------------------------
--              UNARY_LOGICAL_FORMULA
----------------------------------------------------
UNARY_LOGICAL_FORMULA ::= not
                          => as_formula : FORMULA;

not =>;

----------------------------------------------------
--                ATOMIC_FORMULA
----------------------------------------------------
ATOMIC_FORMULA ::= proposition_symbol
                 | result
                 | predicate
                 | true
                 | false
                    | equals;

proposition_symbol => as_symbol : SYMBOL;

result => as_term1 : TERM,
          as_term2 : TERM;

predicate => as_symbol : SYMBOL,
             as_terms : TERM_S;
true =>;
false =>;

equals => as_left_term : TERM,
          as_right_term : TERM;

----------------------------------------------------
--              MODAL_LOGIC_FORMULA
----------------------------------------------------
MODAL_LOGIC_FORMULA ::= believe
                     |  uncertainty
                     |  intention
                     |  persistent_goal
                       => as_agent : TERM,
                         as_formula : FORMULA;

believe =>;
uncertainty =>;
intention =>;
persistent_goal =>;

----------------------------------------------------
--              ACTION_FORMULA
----------------------------------------------------
ACTION_FORMULA ::= done
                |  feasible
                => as_action : TERM,
                    as_formula : FORMULA; -- Added by TM July, 29th, 2004.
done =>;
feasible =>;

----------------------------------------------------
--              QUANTIFIED_FORMULA
----------------------------------------------------
QUANTIFIED_FORMULA ::= exists
                    |  forall
                       => as_variable : VARIABLE,
                         as_formula : FORMULA;

exists =>;
forall =>;

----------------------------------------------------
--             BINARY_LOGICAL_FORMULA
----------------------------------------------------
BINARY_LOGICAL_FORMULA ::= implies
                             | equiv
                         | or
                             | and
                            => as_left_formula : FORMULA,
                                as_right_formula : FORMULA;
implies =>;
equiv =>;
or =>;
and =>;

-----------------------------------------------------
--                    TERM
-----------------------------------------------------
TERM_S ::= Seq of TERM;

TERM ::= VARIABLE
       | CONSTANT
       | TERM_SET
       | TERM_SEQUENCE
       | FUNCTIONAL_TERM
       | ACTION_EXPRESSION
       | IDENTIFYING_EXPRESSION
       | meta_term_reference
       => sm_simplified_term : TERM
{
    TERM getSimplifiedTerm();
    void simplify();
    jade.semantics.lang.sl.tools.MatchResult match(Node expression);
    Node instantiate(String varname, Node expression);
};

meta_term_reference => lx_name : (java.lang.String),
                          sm_value : TERM;

-----------------------------------------------------
--            IDENTIFYING_EXPRESSION
-----------------------------------------------------
IDENTIFYING_EXPRESSION ::= any
                           | iota
                          | all
                              | some
                           => as_term : TERM,
                           as_formula : FORMULA;

any => ;
all => ;
iota => ;
some => ;

-----------------------------------------------------
--                   VARIABLE
-----------------------------------------------------
VARIABLE_S ::= Seq of VARIABLE;

VARIABLE ::= variable
           | meta_variable_reference
           => lx_name : (java.lang.String);

variable => ;

meta_variable_reference => sm_value : VARIABLE;

-----------------------------------------------------
--                   CONSTANT
-----------------------------------------------------
CONSTANT ::= integer_constant
```

```
            | real_constant
            | STRING_CONSTANT
            | date_time_constant
{
          Long intValue();
          Double realValue();
          String stringValue();
};

STRING_CONSTANT ::= string_constant
                  | word_constant
                  | byte_constant;

integer_constant => lx_value : (java.lang.Long);
real_constant => lx_value : (java.lang.Double);
string_constant => lx_value : (java.lang.String);
word_constant => lx_value : (java.lang.String);
byte_constant => lx_value : (byte[]);
date_time_constant => lx_value : (java.util.Date);

----------------------------------------------------
--                    TERM_SET
----------------------------------------------------
TERM_SET ::= term_set;

term_set => as_terms : TERM_S;

----------------------------------------------------
--                   TERM_SEQUENCE
----------------------------------------------------
TERM_SEQUENCE ::= term_sequence;

term_sequence => as_terms : TERM_S;

----------------------------------------------------
--                  ACTION
----------------------------------------------------
ACTION_EXPRESSION ::= action_expression
                    | alternative_action_expression
                    | sequence_action_expression
                   => sm_action : (jade.semantics.actions.SemanticAction)
{
          TERM_S getAgents();
};

action_expression => as_agent : TERM,
                     as_term : TERM;

alternative_action_expression => as_left_action : TERM,
                                 as_right_action : TERM;

sequence_action_expression => as_left_action : TERM,
                              as_right_action : TERM;

----------------------------------------------------
--                 FUNCTIONAL_TERM
----------------------------------------------------
FUNCTIONAL_TERM ::= functional_term
                  | functional_term_param
                  => as_symbol :SYMBOL;

functional_term => as_terms : TERM_S;
```

```
functional_term_param => as_parameters : PARAMETER_S
{
    TERM getParameter(String name);
    void setParameter(String name, TERM term);
};

----------------------------------------------------
--                 PARAMETER
----------------------------------------------------
PARAMETER_S ::= Seq of PARAMETER;

PARAMETER ::= parameter
            => lx_name : (java.lang.String),
              lx_optional : (java.lang.Boolean),
              as_value : TERM;

parameter =>;

----------------------------------------------------
--                   SYMBOL
 ----------------------------------------------------
SYMBOL ::= symbol
         | meta_symbol_reference;

symbol => lx_value : (java.lang.String);

meta_symbol_reference => lx_name : (java.lang.String),
                           sm_value : SYMBOL;
```