The tuPInJADE package

The **tuPInJADE** package defines the following classes:

ErrorMsg: this class is used by the tuProlog agents running in a JADE platform with the aim of displaying a pop-up window with an error/failure message, since the platform does not provide a similar mechanism.

JadeShell42P: this class implements a tuProlog agent and, as the name suggests, it can be considered as the skeleton for any tuProlog agent running in a JADE platform. This agent is ready to be loaded into a JADE platform and a Prolog inference engine (implemented by tuProlog) is already included inside of it. When launching this agent in JADE, it is necessary to input the name of a text file containing the (tu)Prolog theory defining the agent's behaviour. The tuProlog theory specified in that file is loaded into the tuProlog engine and is added to the standard predicates. The life cycle of this agent consists in continuously solving the goal "**main.**" that must be defined in the theory file.

JadeShell42PGui: this class differs from **JadeShell42P** only in the way that the file containing the tuProlog theory defining the agent's behaviour is inputted to it. When loading the agent in JADE, a GUI is displayed from which the user can browse the file system and select the theory file.

TuJadeLibrary: this class implements the tuProlog library (entirely developed in Java) that a tuProlog agent needs in oder to communicate with other agents in any JADE platform. It defines communicative predicates based on the facilities that JADE offers to its agents for communication with other agents running in any JADE platform.

The tuProlog agents

There are 2 ways to load a **JadeShell42P** or **JadeShell42PGui** agent into a Jade platform:

- **from the RMA GUI**

  Click the "**Start New Agent**" item from the "**Actions**" menu in the title bar. A GUI will open and the appropriate parameters regarding the new agent must be inputted. The parameters that the user must input to create a tuProlog agent are: the local name of the agent (chosen by the user) and the fully qualified name of the Java class

implementing the agent (**tuPInJADE.JadeShell42P** or **tuPInJADE.JadeShell42PGui**). For a **JadeShell42P** agent it is also needed to input in the **Arguments** field in the GUI the path of the theory file. For a **JadeShell42PGui** agent, after inputting the name of the agent and the name of the Java class implementing it, a window will open and the user will have to select the theory file, locating it in the system. In both cases, the agent will be created in the selected agent container of JADE.

- **from the command line when launching an agent container**

  To load a **JadeShell42P** agent, just add the following string:

  name:tuPInJADE.JadeShell42P(theory)

  where **name** should be substituted with the name that the user wishes to assign to the agent. The name is local to the platform in which the agent is loaded; the globally unique name that JADE assigns to the agent is a string formed by concatenating this local name, the symbol "**@**" and the platform's identifier, respectively (for more details see the JADE manuals). Finally, **theory** should be substituted with the name (comprising the entire path if the file is not in the current classpath) of the file containing the tuProlog theory defining the **main** predicate.
  More than one tuProlog agent can be specified when launching the container: it is sufficient to separate each string of this form by a blank space (as with non-tuProlog agents).

  To load a **JadeShell42PGui** agent, just add the following string:

  name:tuPInJADE.JadeShell42PGui

  where **name** should be substituted with the name that the user wishes to assign to the agent.
  For each **JadeShell42PGui** agent specified in the command line, a window will open and the user will have to select the theory file locating it in the system.
  Both types of tuProlog agents can be loaded together in a JADE platform.

When this agent is loaded into JADE, its first action is to check if the user has inputted the name of the file that defines the **main** predicate: if the file has not been specified or the system has problems finding/opening/reading it, an error message is displayed by an ad-hoc window and the agent is "killed" (leaving .

If the file can be opened and read by the system, then the agent will check if the theory file begins with "**main:-**" (only lower-case letters are valid in the word **main**) and if it also contains a valid tuProlog theory.

A theory file containing the fact "**main.**" and no other definition of the **main** predicate is considered valid, but the corresponding agent does not ever do anything. If any of the previous checks ends negatively, then the user is made aware of it by the appearance of a window displaying an error message, and the agent "dies".
On the other hand, if these checks end positively, the tuProlog engine is created and, by default, it contains the standard tuProlog libraries (implemented in Java and fully detailed in the "tuProlog Users Guide" available in the documentation provided with the tuProlog system) and the theory inputted when loaded.

After adding the inference engine, the agent tries to extend it with the **tuPInJADE.TuJadeLibrary** (the Java library that we have developed and that defines the predicates for allowing communication between tuProlog agents and other agents in JADE): if it succeeds, the agent can now communicate with any agent running in JADE platforms, otherwise it terminates its life cycle.
A tuProlog theory is represented by text consisting of a series of clauses and/or directives, each followed by "**.**" and the blank space. The theory inputted to the agent "completes" the agent since it determines its **knowledge base** and the **rules by which it accomplishes demonstrations**.

The tuProlog agent is able to carry out logic reasoning when a query is submitted to it, but we have chosen to only submit one query: every time that this agent is scheduled by JADE, the agent automatically proves the "**main.**" goal. If the resolution does not succeed then an error message is displayed to the user.

A typical behaviour of a tuProlog agent may be to read a new message from its messages queue (automatically created by JADE for each agent), handle it and take some action (such as the update of its knowledge and/or message delivery) according to the facts and rules currently present in its theory.
The demonstration process is not visible to the programmer: to see the bindings of the variables made during the resolution, the programmer of the agent has to explicitly write the variables on the standard output or in files.

So, the behaviour of the tuProlog agents is to prove the **main** predicate, but they can differ in the **facts** and **clauses** that rule their reasoning. Comparing them to the other kind of agents runnable in JADE, the tuProlog agents have only one activity to

fulfil, that is the demonstration process of the **main** goal.

The built-in predicates of tuProlog agents

A tuProlog agent has **"built-in" predicates**: the ones provided by the tuProlog libraries and the ones defined in the **TuJadeLibrary** that is automatically loaded into the inference engine of the agent. The **TuJadeLibrary** implements a tuProlog library defining predicates that allow tuProlog agents to **send/receive messages** to/from other agents running in JADE and to **perform specific requests to the default DF** (**D**irectory **F**acilitator) of the JADE platform in which they are running.
The predicates defined in the **TuJadeLibrary** are listed below:

- **ask_address(Role,Address)**
- **reg_role(Role)**
- **send(Performative,Content,Receiver)**
- **send(Performative,Content,Receiver,Protocol,Cid)**
- **receive(Performative,Content,Sender)**
- **rec_cid_role(Performative,Content,Sender,Cid,Role)**
- **blocking_receive(Performative,Content,Sender)**
- **blocking_receive(Performative,Content,Sender,Msec)**
- **pack(Term,StrTerm)**
- **unpack(StrTerm,Term)**

Besides the predicates for sending and receiving messages, there are two predicates for converting strings into **tuProlog terms** and vice versa, a predicate for asking the default DF of the JADE platform in which the agent is running for the address of an agent given the role it registered with the DF, and a predicate to register a role with the default DF.

The ordinary content of a JADE message is a string, even though an object can be used. We have chosen to make the **tuProlog agents send messages whose content is a string** in order to maintain the lightweight mechanism of messaging provided by the JADE framework: this is why we have added the **pack** and **unpack** predicates.
So, on the one hand, when a tuProlog agent has to send a message, the developer includes a call to the **pack** predicate first, and then the call to the **send** predicate.
On the other hand, when a tuProlog agent has to receive a message, the developer includes the call to the **receive**/**rec_cid_role**/ **blocking_receive** predicate and, afterwards, converts the content of the message into a tuProlog term by calling the **unpack** predicate; then, the agent will be able to reason over the received content as it usually does with ordinary terms.

ask_address/2
This predicate is the Prolog counterpart of the Java **search** method (available to any agent running in JADE) for asking the default DF agent for the address of the agent that has registered with it the given role. It takes two arguments: a string representing the role and a string representing a JADE GUID (**G**lobal **U**nique **ID**entifier) of an agent.
The call to the goal **ask_address(Role,Address)** succeeds if, and only if:

- the **Role** term is a string, or a term that unifies with a string, (not the Prolog anonymous variable "_") representing a role;
- the **Address** term is a variable not bound yet.

If the goal is successful, the **Address** term is bound to the GUID of the agent that registered the role **Role** with the default DF, while it remains unbound in case no agent has registered that role. If an error occurs while asking the address to the DF agent, an error message appears to the user.

reg_role/1
This predicate is the Prolog counterpart of the Java **addProtocols** method (available to any agent running in JADE) that allows an agent to register with the default DF a protocol, which is a string. DCaseLP agents use this protocol string to store a role instead. It takes only one argument: a string representing the role that the agent wishes to register with the DF.
The call to the goal **reg_role(Role)** succeeds if, and only if:

- the **Role** term is a string, or a term that unifies with a string, (not the Prolog anonymous variable "_") representing a role.

If the goal is successful, the role **Role** is successfully registered with the DF. If an error occurs while registering the role with the DF agent, an error message appears to the user.

send/3
This predicate is the Prolog counterpart of the Java **send** method (available to any agent running in JADE) for sending a message. To use the send method, the message that the agent wishes to send must be passed to the method as an input parameter. In our case, the message is automatically created and only the performative, the content and the address/addresses of the receiver/receivers are necessary. This predicate takes three arguments that are all strings (the third argument can be a list of strings) used to "complete" the message to send.
The call to the goal **send(Performative,Content,Receiver)** succeeds if, and only if:

- the **Performative** term is a string, or a term that unifies with

a string, representing one of the available performatives of an ACLMessage in JADE (in capital letters). If the string is not a valid performative, the goal still is successful, but the message that is sent has the default performative NOT_UNDERSTOOD;
- the **Content** term is a string, or a term that unifies with a string, (not the Prolog anonymous variable "_") representing the content of the message to send;
- the **Receiver** term is a string, or a term that unifies with a string, representing the address (that is, the GUID) of the agent to which the message is to be sent. The goal is also successful if this term is a list of strings or of terms that unify with strings representing GUIDs.

If the goal is successful, the message is sent to the address(es) specified by the third argument, while the address of the agent sending the message is automatically inserted into the message itself. If the arguments are not valid ones or an error occurs while sending the message, an error message appears to the user.

send/5
This predicate performs the same action of the previous **send** predicate, but it adds more information to the message sent. It takes five arguments that are all strings (the third argument can be a list of strings) used to "complete" the message to send.
The call to the goal **send(Perf,Cont,Rcver,Prot,Cid)** succeeds if, and only if:
- the **Perf** term is a string, or a term that unifies with a string, representing one of the available performatives of an ACLMessage in JADE (in capital letters). If the string is not a valid performative, the goal still is successful, but the message that is sent has the default performative NOT_UNDERSTOOD;
- the **Cont** term is a string, or a term that unifies with a string, (not the Prolog anonymous variable "_") representing the content of the message to send;
- the **Rcver** term is a string, or a term that unifies with a string, representing the address (that is, the GUID) of the agent to which the message is to be sent. The goal is also successful if this term is a list of strings or of terms that unify with strings representing GUIDs;
- the **Prot** term is a string, or a term that unifies with a string, (not the Prolog anonymous variable "_") representing the role of the agent sending the message;
- the **Cid** term is a string, or a term that unifies with a string, (not the Prolog anonymous variable "_") representing the conversation-id of the message that is to be sent.

If the goal is successful, the message is sent to the address(es) specified by the third argument and the address of the agent

sending the message is automatically inserted into the message itself. If the arguments are not valid ones or an error occurs while sending the message, an error message appears to the user.

receive/3
This predicate is the Prolog counterpart of the Java **receive** method (available to any agent running in JADE) for receiving a message. Calling the receive method implies removing a message from the messages queue belonging to the agent and then "reading" the relevant information contained in the message.
This predicate takes three arguments that are all strings that will contain the information read from the message received.
The call to the goal **receive(Performative,Content,Sender)** succeeds if, and only if:

- the **Performative** term is a variable not bound yet;
- the **Content** term is a variable not bound yet;
- the **Sender** term is a variable not bound yet.

If there is at least one message in the queue, it is removed and read, and the goal is successful. The performative, the content and the address of the sender are bound to the arguments of the goal, respectively. If, on the contrary, the queue is empty, the goal still succeeds, but its arguments remain unbound variables.
If an error occurs while receiving the message, an error message appears to the user.

rec_cid_role/5
This predicate performs the same action of the previous **receive** predicate, but it provides more information about the message received. It takes five arguments that are all strings that will contain the information read from the message received.
The call to the goal **rec_cid_role(Perf,Cont,Sdr,Cid,Role)** succeeds if, and only if:

- the **Perf** term is a variable not bound yet;
- the **Cont** term is a variable not bound yet;
- the **Sdr** term is a variable not bound yet;
- the **Cid** term is a variable not bound yet;
- the **Role** term is a variable not bound yet.

If there is at least one message in the queue, it is removed and read, and the goal is successful. The performative, the content, the address of the sender, the conversation-id and the protocol (used by the sender to specify its role) are bound to the arguments of the goal, respectively. If, on the contrary, the queue is empty, the goal still succeeds, but its arguments remain unbound variables.
If an error occurs while receiving the message, an error message appears to the user.

blocking_receive/3
This predicate is the Prolog counterpart of the Java **blockingReceive** method (available to any agent running in JADE) for blocking the agent in case its messages queue is empty. If the queue is empty, the agent will wait for a message to arrive, otherwise it removes a message from the queue and reads it. This predicate takes three arguments that are all strings that will contain the information read from the message received. The call to the goal **blocking_receive(Perf,Content,Sender)** succeeds if, and only if:

- the **Perf** term is a variable not bound yet;
- the **Content** term is a variable not bound yet;
- the **Sender** term is a variable not bound yet.

When a message is present in the queue, it is removed and read, and the goal is successful. The performative, the content and the address of the sender are bound to the arguments of the goal, respectively. If an error occurs while receiving the message, an error message appears to the user.

blocking_receive/4
This predicate performs the same action of the previous **blocking_receive** predicate, but it constraints the period of time for which the agent will block while waiting for a message to arrive. It takes four arguments, the first three are strings that will contain the information read from the message received, while the last one is a number specifying the maximum amount of milliseconds for which the agent will block. The call to the goal **blocking_receive(Perf,Cont,Sdr,Msecs)** succeeds if, and only if:

- the **Perf** term is a variable not bound yet;
- the **Cont** term is a variable not bound yet;
- the **Sdr** term is a variable not bound yet;
- the **Msecs** term is a number, or a term that unifies with a number, that constraints the period of time for which the agent will block.

When a message is present in the queue, it is removed and read, and the goal is successful. The performative, the content and the address of the sender are bound to the arguments of the goal, respectively. If, on the contrary, the queue is empty, the agent will wait for a message to arrive for the amount of milliseconds specified by the fourth argument of the goal. If an error occurs while receiving the message, an error message appears to the user.

pack/2
This predicate has the aim of **converting a tuProlog term into a string** in order to be able to send that term as the content of a

message. It should be evaluated **before** the **send** predicate.
It takes two arguments that are both strings.
The call to the goal **pack(Term,StrTerm)** succeeds if, and only if:

- the **Term** term is a ground tuProlog term or a variable that unifies with a ground term;
- the **StrTerm** term is a variable not bound yet.

If the goal is successful, the **StrTerm** term is bound to the string representation of the **Term** term. If an error occurs while parsing the **Term** term or bounding its representation to the second argument, an error message appears to the user.

unpack/2
This predicate has the aim of **converting a string into a tuProlog term** in order to be able to reason on the content of a message received. It should be evaluated **after** one of the predicates by which an agent can **receive** a message.
It takes two arguments that are both strings.
The call to the goal **unpack(StrTerm,Term)** succeeds if, and only if:

- the **StrTerm** term is a string, or a term that unifies with a string, representing a tuProlog term;
- the **Term** term is a variable not bound yet.

If the goal is successful, the **Term** term is bound to the tuProlog term represented by the **StrTerm** string. If an error occurs while parsing the **StrTerm** term or bounding the tuProlog term that it represents to the second argument, an error message appears to the user.