

1. Providing the high level specification of the MAS

The UMLInJADE package provides the means to translate, in a semi-automatic way, the high level specification of the MAS, given either in UML or in an XML intermediate format, into skeletons of Jess agents. We need to define the format of these high level specification languages before explaining how the UMLInJADE package works.

1.1 Specifying protocol diagrams

Protocol diagrams set the interaction rules among roles that will be played by classes of agents. From now on, we will write that a role sends (receives) a message, to mean that the agents that are instances of those classes that play that role, are able to send (receive) the message.

There is only one means to specify protocol diagrams, namely, using the XML intermediate format. The DTD of protocol diagrams in XML intermediate format is the following:

```
<!Element protocoldiagram (role+) >
<!Element role (msgs) >
<!Element msgs (msg+, xor-send+, or-send+, and-send+, xor-thread+, or-thread+,
and-thread+) >
<!Element msg (sender,receiver,act) >
<!Element xor-send (sender, receiver, act+) >
<!Element or-send (sender, receiver, act+) >
<!Element and-send (sender, receiver, act+) >
<!Element xor-thread (thread+) >
<!Element or-thread (thread+) >
<!Element and-thread (thread+) >
<!Element thread (msg+, xor-send+, or-send+, and-send+, xor-thread+, or-thread+,
and-thread+) >
<!Element sender (#PCDATA) >
<!Element receiver (#PCDATA) >
<!Element act (#PCDATA) >
```

For each role that has been identified during the analysis stage (**role** element in the DTD), the set of sent and received messages must be specified (**msgs** element in the DTD). For each role, both the messages sent, and those received, must be specified. A message is sent by the role R, if the role that sends the message (**sender** element in the DTD) is equal to the role R. A message is received by a role R, if the role that receives the message (**receiver** element in the DTD) is equal to the role R. Sent messages, may be either simple messages (**msg** element in the DTD, characterised by its sender, its receiver, and the performative of the message – **act** element), or complex messages (**xor-send**, **or-send**, **and-send** elements in the DTD, each one characterised by sender, receiver and a set of performatives). Receiving a complex message may split the main thread of the receiving role into more threads. For example, the reception of three mutually exclusive messages (sent using a **xor-send**), requires a splitting of the thread into three different threads; in each of the three threads, one of the messages sent by means of the **xor-send**, may be received. The **xor-thread** element allows this kind of splitting due to the reception of a **xor-send** message. In a

similar way, a set of messages sent using an or-send, or an and-send, may be received using an **or-thread**, or an **and-thread**, respectively.

As an example, we may consider the following protocol diagram, that corresponds to the FIPA propose protocol (<http://www.fipa.org/specs/fipa00036/SC00036H.html>) shown in Figure 1.

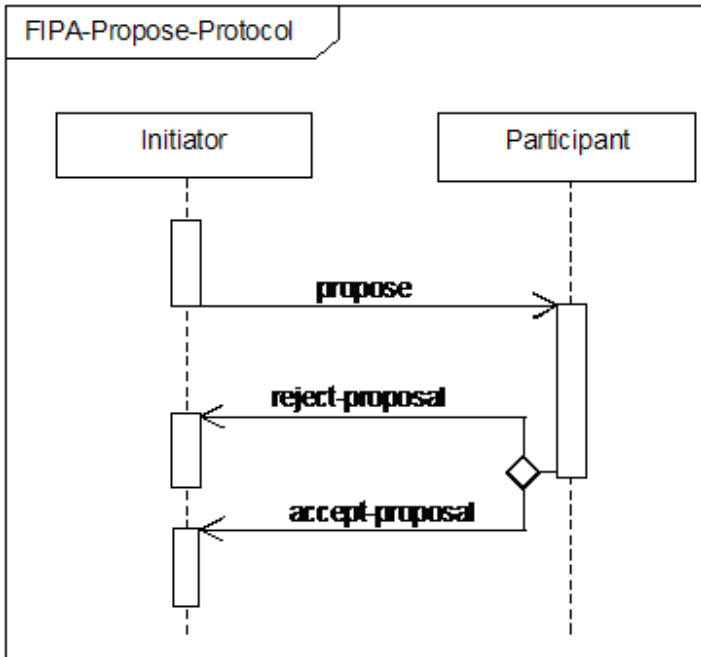


Figure 1: FIPA-Propose-Protocol

```

<protocoldiagram>
<role>
<name>Seller</name>
<msgs>
<msg>
<sender>Seller</sender>
<receiver>Buyer</receiver>
<act>PROPOSE</act>
</msg>
<xor-thread>
<thread>
<msg>
<sender>Buyer</sender>
<receiver>Seller</receiver>
<act>REJECT_PROPOSAL</act>
</msg>
</thread>
<thread>
<msg>
<sender>Buyer</sender>
<receiver>Seller</receiver>
<act>ACCEPT_PROPOSAL</act>
</msg>
</thread>
  
```

```

</xor-thread>
</msgs>
</role>
<role>
<name>Buyer</name>
<msgs>
<msg>
<sender>Seller</sender>
<receiver>Buyer</receiver>
<act>PROPOSE</act>
</msg>
<xor-send>
<sender>Buyer</sender>
<receiver>Seller</receiver>
<act>REJECT_PROPOSAL</act>
<act>ACCEPT_PROPOSAL</act>
</xor-send>
</msgs>
</role>
</protocoldiagram>

```

1.2 Specifying architecture diagrams

Architecture diagrams express the relationships that exist between roles and classes of agents. They can be expressed either by means of a UML class diagram, or in XML intermediate format.

By means of UML class diagrams

UML class diagrams representing the architecture of the MAS (namely, which classes of agents play which role) have the structure shown in Figure 2.

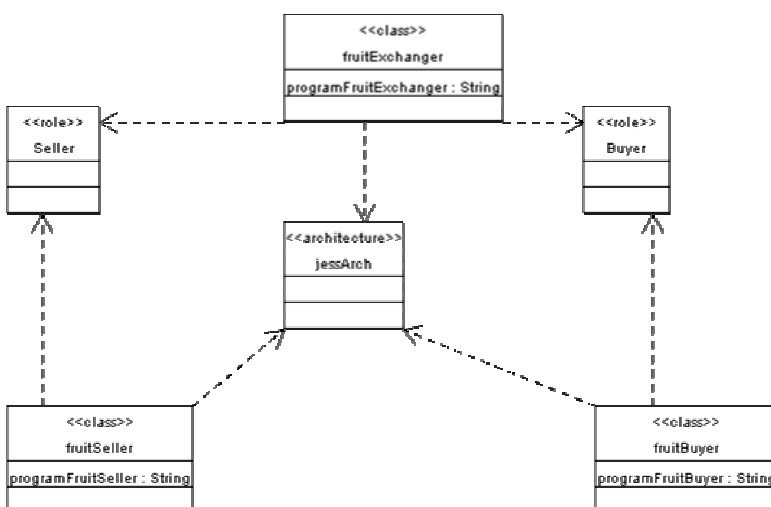


Figure 2: DCaseLP architecture diagram

Each agent class, characterised by the <<class>> stereotype, has an attribute that, during the translation process, will be used to identify the name of the Jess programs for that class. Each agent class is related to a role class (class with stereotype <<role>>) by a dependency relation named `plays_role`.

A class with stereotype <<architecture>> is used to specify which kind of agent architecture (BDI, reactive, etc) characterises the agent class. For the moment, only one architecture (`jessArch`, meaning that the agents are rule-based declarative agents whose behaviour is described by means of jess rules) is implemented, and for this reason, the information on the agent architecture is not translated into the intermediate code (it is a useless information, since only one architecture can be chosen). The dependency relation between the agent class and the architecture class is named `has_architecture`.

By means of the XML intermediate format

The DTD of the architecture diagram, expressed in XML intermediate format, is the following:

```
<!Element architecturediagram (class+) >
<!Element class (name,program,roles) >
<!Element roles (role+) >
<!Element name (#PCDATA) >
<!Element program (#PCDATA) >
<!Element role (#PCDATA) >
```

Here, elements have their intuitive meaning, so they do not need much explanation. Instead, we show the XML intermediate format for the diagram shown in Figure 2.

```
<architecturediagram>
<class>
<name>fruitSeller</name>
<program>programFruitSeller</program>
<roles>
<role>Seller</role>
</roles>
</class>
<class>
<name>fruitBuyer</name>
<program>programFruitBuyer</program>
<roles>
<role>Buyer</role>
</roles>
</class>
<class>
<name>fruitExchanger</name>
<program>programFruitExchanger</program>
<roles>
<role>Seller</role>
<role>Buyer</role>
</roles>
</class>
</architecturediagram>
```

1.3 Specifying agent diagrams

By means of UML class diagrams

Agent diagrams state which agent classes have which instances. The agent classes already defined in the architecture diagram must be included in the agent diagram (here, the stereotype is `<<agentclass>>`) together with their instances (with stereotype `<<agent>>`, and with one attribute named `state`). The dependency relation between instances of agents, and agent classes, is named `instance_of`.

For example, an agent diagram that defines the agent instances of the agent classes shown in Figure 2, is shown in Figure 3.

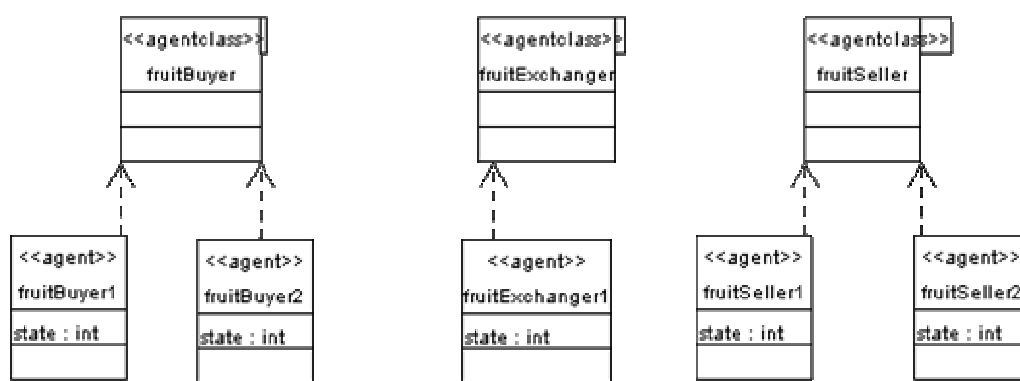


Figure 3: DCaseLP agent diagram

By means of the XML intermediate format

The DTD of the architecture diagram, expressed in XML intermediate format, is the following:

```
<!Element agentdiagram (agent+) >  
<!Element agent (name,class,state) >  
<!Element name (#PCDATA) >  
<!Element class (#PCDATA) >
```

Again, the meaning of the elements is easy to understand, and, instead of explaining the DTD in de tail, we provide the representation, in XML intermediate format, of the agent diagram shown in Figure 3.

```
<agentdiagram >  
<agent >  
<name>fruitBuyer1 </name >  
<class>fruitBuyer </class >  
</agent >  
<agent >  
<name>fruitBuyer2 </name >  
<class>fruitBuyer </class >  
</agent >  
<agent >  
<name>fruitSeller1 </name >
```

```

<class>fruitSeller</class>
</agent>
<agent>
<name>fruitSeller2</name>
<class>fruitSeller</class>
</agent>
<agent>
<name>fruitExchanger1</name>
<class>fruitExchanger</class>
</agent>
</agentdiagram>

```

2. Translating the high level specification: the UMLInJADE package

The **UMLInJADE** package defines the following class:

Specif2Code: this class is used to perform the translation from the high level description of the MAS (either given by means of UML, exported into XMI, or by means of the intermediate XML format) into skeletons of Jess agents that behave according to the interaction protocols, and are organised according to the architecture and agent diagrams. The Java “stubs” necessary to integrate the Jess code into JADE are also automatically generated. Specif2Code uses two auxiliary classes, namely Selector and Specif2Code\$Handler.

The translation is performed by the XSL processor to process the style sheets contained into the DCASELP\UMLInJADE\XSL directory. In particular,

- the xmiarchitecture.xml style sheet translates the architecture diagram (in XMI format) into the intermediate XML format.
- the xmiagent.xml style sheet translates the agent diagram (in XMI format) into the intermediate XML format.
- the xmitrader.xml style sheet is used when the XMI models are all in one single file.
- the xmltrader.xml style sheet translates the architecture, agent and protocol diagrams (in intermediate XML format) into the corresponding Java and Jess skeletons (one for each class of agent, as specified in the architecture diagram).

The Java code produced by this translation is ready to be run into the JADE platform, but the Jess code that characterises the behaviour of each agent must be completed by the developer, in order to contain all the information required for the agent's functioning. In fact, in the high level specification of the MAS described in Section 1, some details necessary for running the simulation are missing. In particular, the interaction protocol specifies neither under which conditions one message should be sent, nor which actions (apart from sending and receiving messages) the agent should take after a communicative action. These conditions and these actions must be added to the generated Jess code by the developer. Also the initial state of the agent should be added to the Jess code.

The usage of UMLInJADE.Specif2Code is simple: from a shell, just type in “java UMLInJADE.Specif2Code” and enter the information that is required by means of a set of interactive windows.

- A first window asking to select a file from the XSL directory appears. Select any file from the DCASELP\UMLInJADE\XSL directory (note that this is a quick way to select the entire XSL directory, not the specific file, so it does not matter which file are you choosing, provided that it is in the right directory).
- After that, a window asking if there are XMI diagrams in the specification of the MAS appears. The answer may be yes or not, depending on whether the diagrams have been defined using UML, and then exported into XMI, or they have been directly defined using the intermediate XML format.
- Then, a window asking how many protocol diagrams there are in the system appears; type the number of protocol diagrams, then press the DONE button, and then select the protocol diagram in XML intermediate format.
- In a similar way, a window asking if there is an XMI architecture diagram appears, and after that, a window asking if there is an XMI agent diagram appears; the answer may be yes or not, depending on the format used for architecture and agent diagrams.

Once all the files corresponding to the high level specification of the MAS have been selected for the translation, the program translates them, and terminates. The output of the translation can be found in the directories UMLInJADE\jacode (the Java stubs for each agent class), UMLInJADE\jocode (the Jess behaviour for each agent class), and UMLInJADE\intermed (the XML intermediate format generated for those diagrams that were initially defined in XMI).

3. Executing the resulting MAS in JADE

3.1 Completing the Jess code

In order to execute the MAS resulting from the translation of the high level specification, the Jess code must be first be completed. The Jess code is characterised by an initial part that defines the functions available to the Jess agent (see the jessInJADE-Manual, in the jessInJADE directory, for an explanation of the functions available to the Jess agent), followed by the rules that implement the interaction protocol defined when specifying the MAS.

An example of generated rules is the following, where pink pieces of code are those that the developer must (eventually) replace with domain-dependent conditions and actions. Strings that begin with “;”, are Jess comments.

```
; ***** RULES *****
(defrule Seller_1 ; additional conditions
=> (bind ?cid (newcid)) (bind ?*addr* (read_addr "Buyer"))
(if (eq nil ?*addr*) then (bind ?*addr* (fetch_addr "Buyer"))))
(bind ?content CONTENT OF THE MSG TO SEND)
(send (ACLMessage (communicative-act PROPOSE)
(receiver ?*addr*) (protocol "Seller") (conversation-id ?cid)
(content ?content))) (assert (state Seller_1 ?cid))
)
```

```

(defrule Buyer_1 ; additional conditions
=> (wait_msg "PROPOSE" "Seller")
(assert (state Buyer_1 ?cid))
; additional actions
(retract-string "(message \"PROPOSE\" \"Seller\")))

(defrule Buyer_2_1
(state Buyer_1 ?cid)
; conditions
=> (bind ?content MSG CONTENT)
(send (ACLMessage (communicative-act REJECT_PROPOSAL) (sender
Buyer) (receiver Seller) (conversation-id ?cid) (content ?content)))
(assert (state Buyer_2_1 ?cid))
(retract-string (str-cat "(state Buyer_1 " ?cid ")"))
; actions)

```

....

In this piece of code, that belongs to the program of the `fruitExchanger` agent class of our running example, the conditions under which messages should be sent, as well as the content of the messages themselves, and the actions to do, besides the communicative ones, must be added. Also the initial knowledge of the agents belonging to the class must be manually added to the “MAIN” section of the file.

Refer to the Jess manual and the UMLInJADE Tutorial for more information on Jess and on its use inside DCaseLP.

3. 2 Integrating the completed Jess agents into JADE

Once all the Jess agents have been completed, the JADE MAS can be built and its simulation can start. Let us suppose that the agent diagram defined in the beginning (see Section 1.3) says that there are three instances of agents of class `classC1`, named `instC11`, `instC12` and `instC13` respectively, and two instances of agents of class `classC2`, named `instC21` and `instC22` respectively.

- First of all, the Java stubs must be compiled in order to obtain the Java classes corresponding to the agent classes. This can be done by typing `javac *.java` from the `jacode` directory.
- Then, always from the `jacode` directory, type `java -gui jade.Boot instC11:classC1 instC12:classC1 instC13:classC1 instC21:classC2 instC22:classC2`

Note that you must issue the above commands from the `jacode` directory, otherwise you will experience some problems. Also note that the Jess code, after completion, must be kept into the `jecode` directory (where it was put by the translator program) and the names of the files cannot be changed.

Now, the JADE GUI should appear, and your five agents should have been loaded into the JADE main container. From now on, you should refer to the JADE Manual for details. Some information on how following the interactions between Jess agents (after their integration into JADE) can also be found in the UMLInJADE Tutorial.