INGENIAS Development Kit (IDK) Manual

Facultad de Informática
Universidad Complutense de Madrid
C/ Prof. JGarcía Santesmases s/n
28040 Madrid
España

http://grasia.fdi.ucm.es

IDK MANUAL

SUMMARY

THIS DOCUMENT PRESENTS THE INGENIAS DEVELOPMENT KIT (IDK), A SET OF TOOLS FOR THE SPECIFICATION, VERIFICATION, AND IMPLEMENTATION OF MULTI-AGENT SYSTEMS. IT IS ORIENTED TO RESEARCHERS LOOKING FOR AN AGENT-ORIENTED SPECIFICATION TOOL AND DEVELOPERS WANTING TO FOLLOW AN AGENT-ORIENTED PROCESS (SUCH AS THE INGENIAS METHODOLOGY).

AS IT IS DISTRIBUTED, THE IDK CAN BE DIRECTLY USED TO SPECIFY, USING A GRAPHICAL EDITOR, MULTI-AGENT SYSTEMS. ITS FUNCTIONALITY CAN BE EXTENDED WITH MODULES THAT PERFORM AUTOMATIC CODE GENERATION AND VERIFICATION OF THE SPECIFICATIONS. SOME OF THESE MODULES ARE PROVIDED WITH THE IDK DISTRIBUTION, BUT THE DEVELOPER CAN ALSO CREATE NEW MODULES FOR A PARTICULAR APPLICATION (E.G., FOR OTHER AGENT PLATFORMS THAN THOSE CURRENTLY SUPPORTED BY IDK). THIS FEATURE IS SUPPORTED BY FACILITIES TO ACCESS THE IDK FRAMEWORK (AN API FOR PROGRAMMING MODULES AND MODULE GENERATION TOOLS), WHICH ARE BASED ON A META-MODEL SPECIFICATION OF MAS, AS DEFINED IN THE INGENIAS PROJECT.

PROJECT: INGENIAS

VERSION: 2.5.2 (Corresponds to version 2.5 of IDK)

STATUS: DRAFT

DATE: 26/11/2005

AUTHORS: Jorge J. Gómez Sanz and Juan Pavón

INDEX

1. INTRODUCTION

This document describes the INGENIAS Development Kit (**IDK**), a set of tools for the specification, verification and implementation of Multi-Agent Systems (MAS). There are two main types of IDK tools:

- *A Graphical Editor*. This allows the developer to create and modify specifications of a multi-agent system (MAS) using agent concepts. The underlying agent model is defined in the INGENIAS methodology, but it is incorporating other languages such as Agent UML. Graphical specifications can be drawn as UML diagrams or using the notation defined in the INGENIAS methodology.
- *Modules.* They allow working with MAS specifications to perform verification of properties and automatic code generation. Apart of the modules that are already provided with the IDK distribution, developers can extend IDK functionality by creating their own modules and plugging them in the IDK to perform verification of specific properties or automatic code generation for a particular target platform. The implementation of modules is supported by a framework that facilitates traversing specifications and producing some output, which can be code or the result of a verification of some properties.

The INGENIAS methodology guides the developer in the process of analysing, designing and implementing a MAS. Although the IDK has been conceived to support the INGENIAS methodology, it can be also used with other development process model in practice.

1.1. SCOPE AND USE

This document provides an introduction to the use of the IDK, and it can be used as a user's manual for the IDK. The specification of the meta-models that are supported by the IDK is available at http://grasia.fdi.ucm.es/ingenias/metamodel. Meta-models provide a detailed specification of the concepts underlying IDK and they should be completely understood by those developers willing to create new modules for IDK.

The information contained in this document is property of the *grasia!* research group, so if you use the information contained here or the IDK software, we would appreciate if you include a reference to it in your papers. You can take the following as an example:

> Gómez-Sanz, J. and Pavón, J. (2005). *INGENIAS Development Kit (IDK) Manual, version 2.5*. Facultad de Informática, Universidad Complutense Madrid, Spain. Available at *http://ingenias.sourceforge.net*

And email us in order to update our referred paper lists and to provide comments. This is important to us in order to improve the tools and the methodology.

Information on the INGENIAS methodology is available at several papers, for instance:

> Pavón J., and Gómez-Sanz J. (2003). Agent-Oriented Software Engineering with INGENIAS. In: *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03),* Lecture Notes in Computer Science 2691, Springer Verlag, pp. 394-403

> Pavón J., Gómez-Sanz J. and Fuentes, R. (2005). The INGENIAS Methodology and Tools. In: Henderson-Sellers, B. and Giorgini, P., editors (2005). *Agent-Oriented Methodologies*. Idea Group Publishing, chapter IX, pp. 236—276

1.2. CREDITS

To the members of the Grasia! Research group, specially to the authors of this technical report (Jorge J. Gómez-Sanz and Juan Pavón). Also to the rest of the people that has collaborated in

the development of the IDK (Rubén Fuentes), and those who participated as beta-testers (Guillermo Jiménez, Juan Antonio Recio, Carlos Celorrio, Alberto Fernández, and many others).

This work has been developed in the project INGENIAS (TIC2002-04516-C03-03), which is funded by Spanish Council for Science and Technology.

1.3. GUIDE TO READ THE DOCUMENT

The document is oriented to two types of readers. Those who just want to use IDK to specify MAS and generate code should go through chapters 1 to 4. Next chapters are more oriented to advanced users who want to create their own modules, for instance, to generate code in other platforms than those supported by the IDK distribution.

Chapter 2 describes how to start working with IDK, by indicating how to install IDK and how to use one of the examples that come with the IDK distribution in order to generate code and execute on a JADE platform. Chapter 3 explains how to use the editor, and some hints and tricks to make your specification work easier. Chapter 4 is dedicated to the specification of AUML diagrams.

For advanced users, chapter 5 describes how to create new modules to extend the functionality of the IDK. It describes how to traverse diagrams and generate code or build a customized code generator.

Chapter 6 can be useful for all readers as it illustrates the use of IDK with several case studies.

Finally, chapter 7 overviews the INGENIAS software process, which can serve as a guide to the development of a MAS.

2. Getting started with the IDK

This section describes how to install the IDK and a simple example of specification of a system with two agents (one of them a mobile agent), their automatic implementation, and deployment on two JADE systems (JADE is a FIPA compliant agent platform, http://jade.tilab.com/).

2.1. Installing the IDK

Currently, the IDK is distributed from http://ingenias.sourceforge.net in form of a zip file. You have to download it and unzip it into your favourite folder, say c:\idk or /home/user/idk.

To run the last version of IDK we recommend **JDK 1.5** or higher. Lower versions do not have some enhancements that newer versions have, like regular expressions handling or clipboard transfer utilities.

To test the install, you may want to start the editor, the main component of the IDK. If you are in the *idk* dir, it can be run directly with:

```
idk> java –jar lib/ingeniaseditor.jar
```

However, we recommend to use the *ant* tool instead. ant can be downloaded from *http://ant.apache.org*. This tool not only starts the editor, it also allows to recompile the editor, compile attached modules, run tests, and other interesting features for developers, specially.

Concrete *ant* install instructions can be found at *http://ant.apache.org/manual/installlist.html*. By using *ant*, the IDK is started with the command:

```
idk> ant runide
```

Where *ant* is the script that launches the *ant* system. In order to make it work, you need to have installed *ant*, and the *ant.bat* or *ant.sh* script in the *PATH* environment variable.

It is also possible to create some command file to make easier the launching of the IDK. For instance, in Windows, edit a new file with the following (assuming that the IDK has been unzipped on c:\idk and that ant is accessible with the system environment PATH variable):

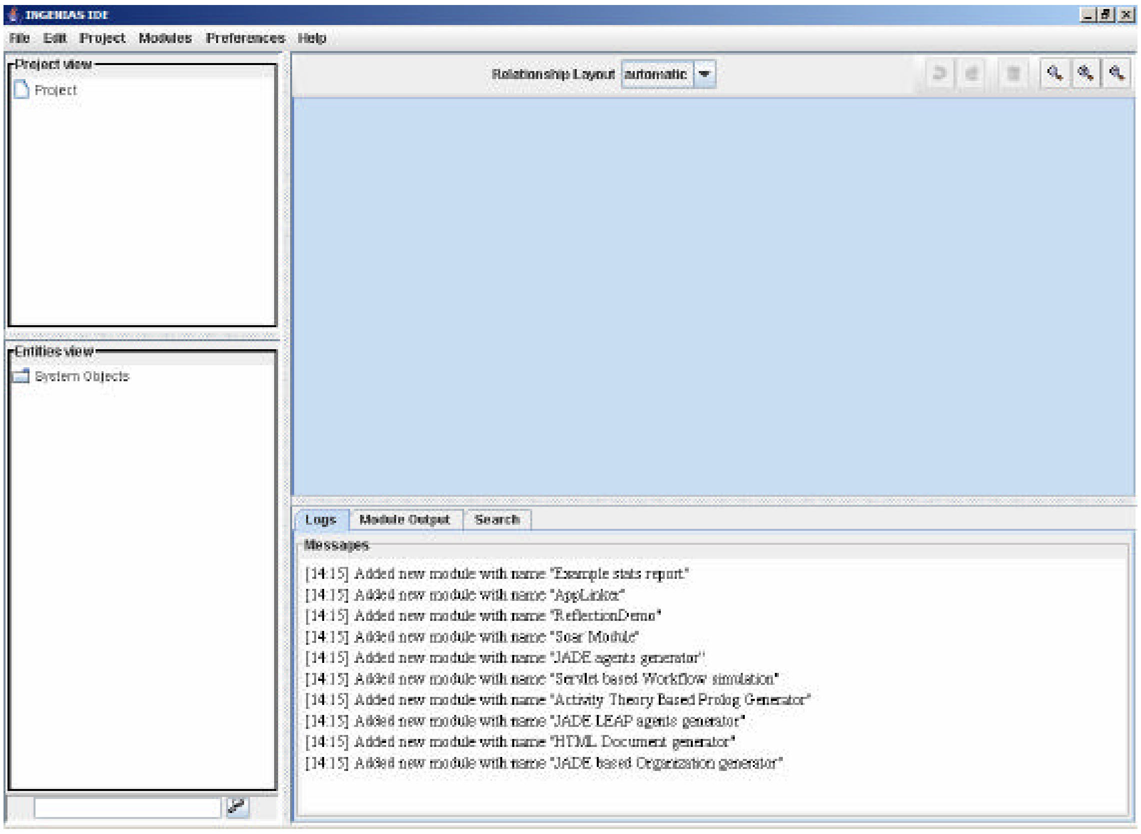
```
c:
cd \idk
ant runide
```

Save the file with this lines as something like IDK.BAT and this can be used to quick launching of IDK.

2.2. Specification of a multi-agent system

As an example of use of IDK we can try one of the examples that are distributed with the IDK. The example is quite simple just in order to illustrate basic functionality of IDK. It has been originally developed by one of our students, Carlos Celorrio, in 2004. Although it does not show all features that can be specified in a multi-agent system, it provides enough information to get some agents finally running on an existing agent platform.

First, start the IDK, and the main editor will appear as shown in Figure 1. Some messages appear in the Logs tab to indicate that several modules of the IDK have been loaded.


INGENIAS IDE
File Edit Project Modules Preferences Help
Project view
Project
Entities view
System Objects
Relationship Layout automatic
Logs Module Output Search
Messages

```
[14:15] Added new module with name "Example stats report"
[14:15] Added new module with name "AppLinker"
[14:15] Added new module with name "ReflectionDemo"
[14:15] Added new module with name "Soar Module"
[14:15] Added new module with name "JADE agents generator"
[14:15] Added new module with name "Servlet based Workflow simulation"
[14:15] Added new module with name "Activity Theory Based Prolog Generator"
[14:15] Added new module with name "JADE LEAP agents generator"
[14:15] Added new module with name "HTML Document generator"
[14:15] Added new module with name "JADE based Organization generator"
```

Figure 1. IDK main screen

To start working, you can create a new project or load one specification. Here, to get the specification of the first example, go to the File menu and select Load (see Figure 2). Then select the specification file for the example. This is in the same directory of the installation of IDK, in a directory called “examples”. There are several examples, and in this case we choose “cinema.xml” (specifications are stored as XML files), as it shows Figure 3.

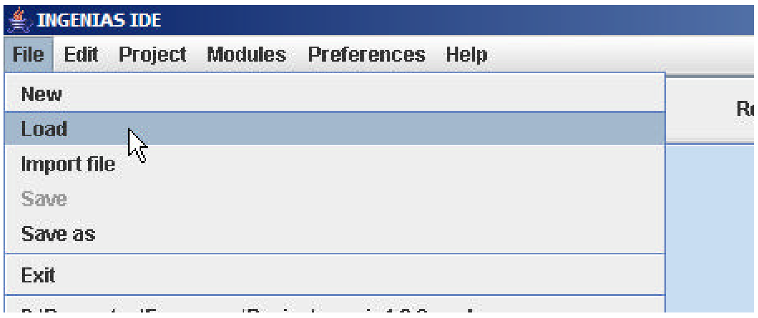


Figure 2. Loading a specification

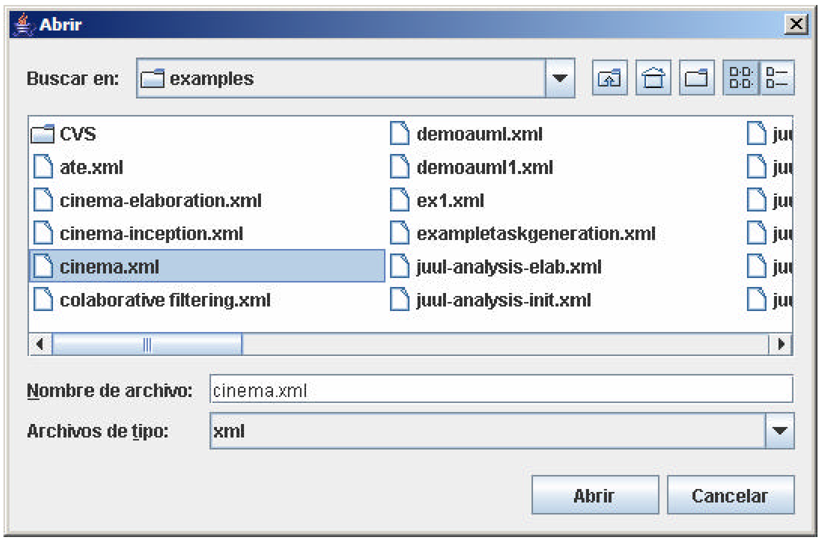


Figure 3. Selecting the cinema specification

When the specification has been loaded, on the left side (Figure 4) we can see the Project view, to navigate through the structure of the specification and the Entities view to navigate through the specification entities. To open a diagram, just click twice on it.

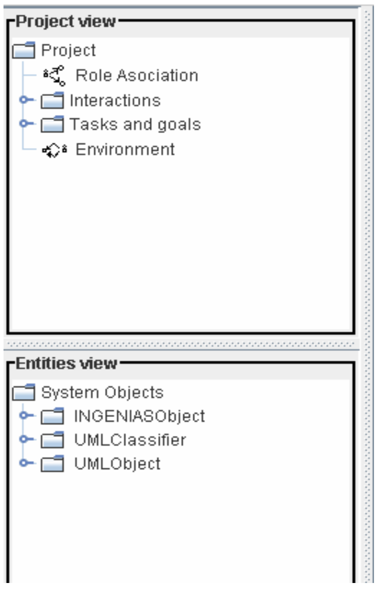


Figure 4. Project and Entities views

In this example, there are three agents. One is an interface agent that interacts with the user in order to get requests for some service and presenting the results. In this case the service is obtaining a ticket for the cinema. The interface agent uses services from other agents. In this example, there is one agent, BuyerAgent, that takes care of finding and buying the cinema tickets for the user. Finally, the SellerAgent has tickets for sale. Each agent in this system plays a role, as specified in the Role Association diagram, which is a diagram of type Agent Model (Figure 5).

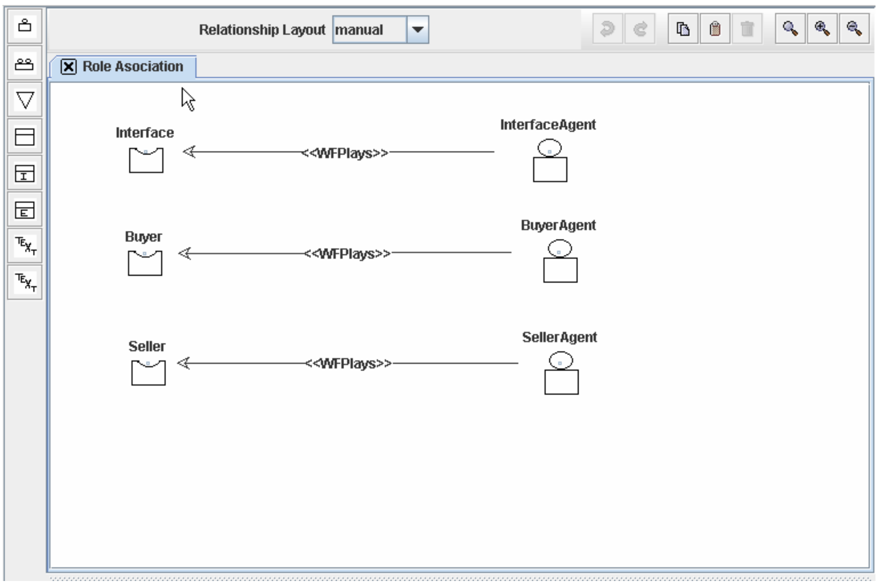


Figure 5. Role Association diagram

The InterfaceAgent and the BuyerAgent initially reside in the PDA environment. The SellerAgent is on another environment, CinemaEnvironment. This can be represented with an environment model diagram, as in Figure 6. Here agents are associated to each application environment with relationship of type <<ApplicationBelongsTo>>.

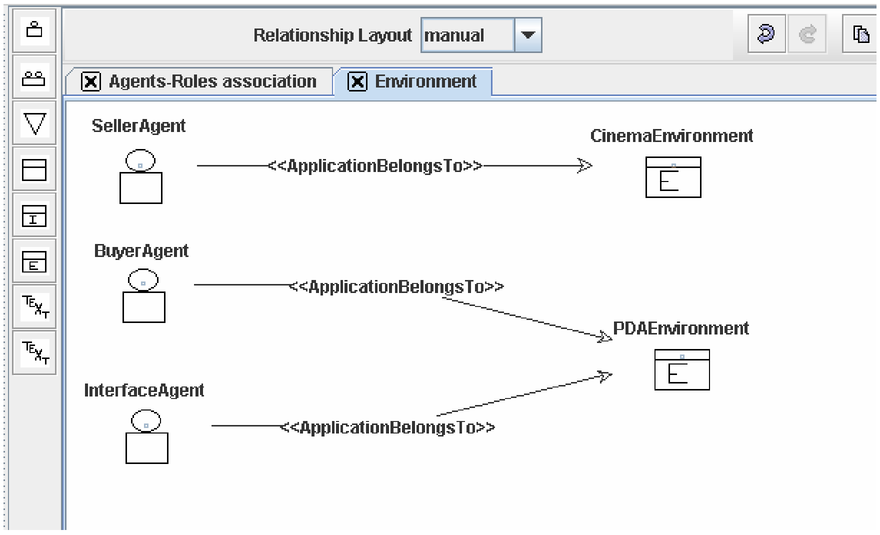


Figure 6. Environment diagram

The behaviour is defined by the goals and tasks of the agents, and the interactions between agents.

The BuyerAgent, for instance, pursues the goal of serving requests for buying, in this case, cinema tickets. This goal can be broken down into simpler goals, such as FindCinema and GetTicket. These simpler goals can be achieved by some plans, sequences of tasks. For instance, to get a ticket the BuyerAgent can choose a cinema, move to that cinema, buy the tickets there, and return to the user’s PDA. This is illustrated in the goals-tasks diagram in Figure 7.

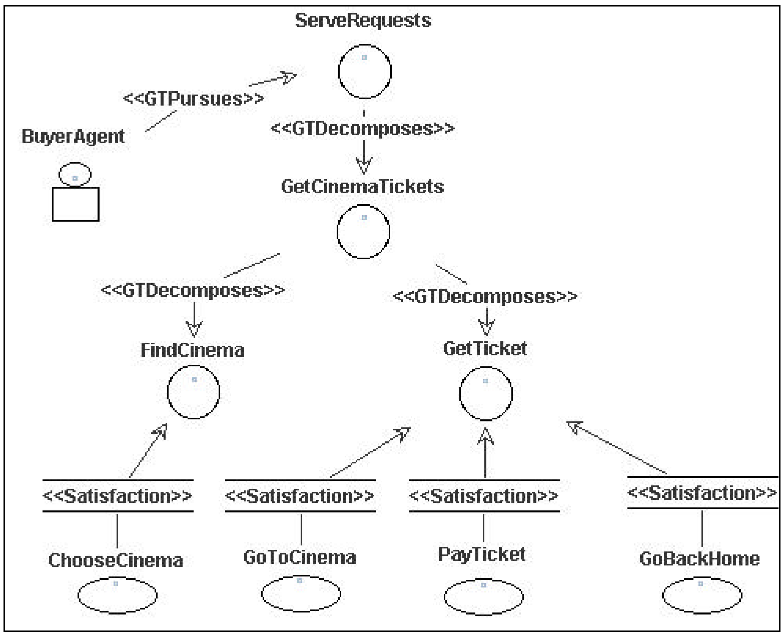


Figure 7. Goals tasks diagram for the Buyer Agent

In this example there is only one interaction, which is shown in Figure 8. In INGENIAS one interaction may be a multi-party association, with one initiator and one or more collaborators, as in this example. In fact, an interaction in INGENIAS is considered something more than just a message-passing and may represent more the idea of a transaction, which may consist of the exchange of several messages or annotations in a shared tuple space, depending on the communication paradigm.

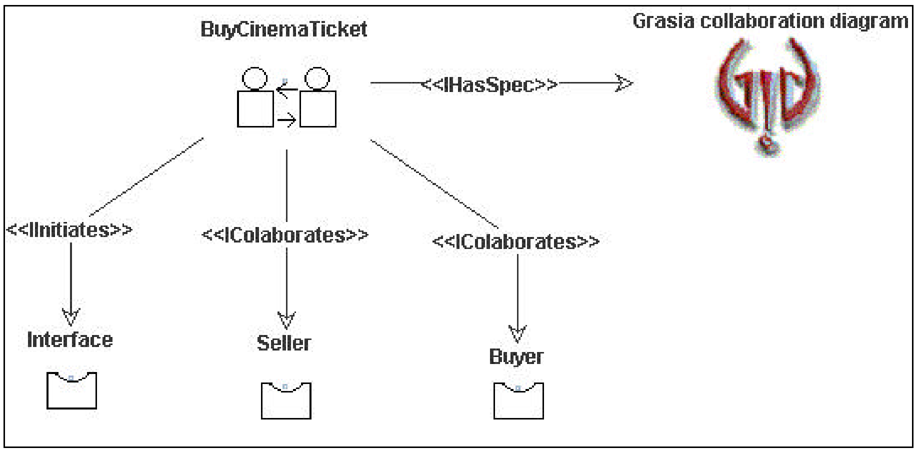


Figure 8. Interaction for buying a ticket

The flow of messages in the specification can be specified by a Grasia collaboration diagram as the one in Figure 9, or by other types of interaction diagrams, such as UML or Agent UML sequence diagrams, see chapter 224. In INGENIAS interactions are triggered in tasks. The flow of tasks and interactions determines the global behaviour of the system. This information is very important when generating code.

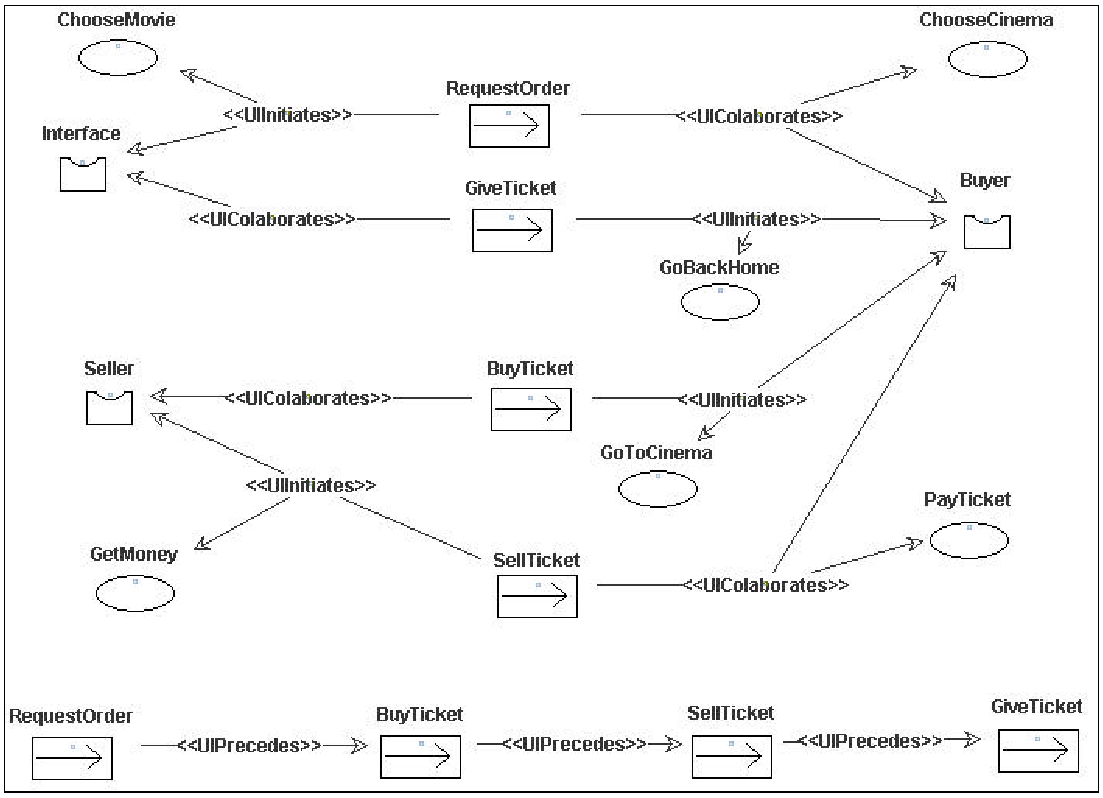
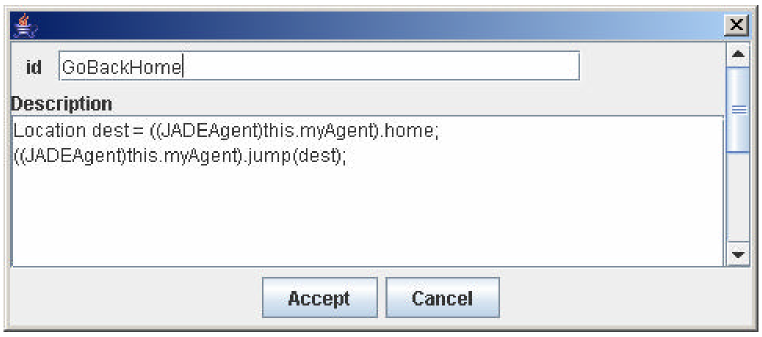


Figure 9. Grasia collaboration diagram

For some tasks, it may be interesting to include some code that can be applied in the implementation. This is the case for the code for mobility in task *GoBackHome*, for instance, that is the code we will need later on for the Jade platform. To get a description of the task, just click on it and you get a window with the identifier for the task and a description field, as in Figure 10, with some code that the code generator module can use.


id GoBackHome

Description

```
Location dest = ((JADEAgent)this.myAgent).home;
((JADEAgent)this.myAgent).jump(dest);
```

Figure 10. Code for task GoBackHome

2.3. CODE GENERATION FOR THE JADE PLATFORM

Once the specification is complete, it is possible to apply the code generation module to implement the agents in some target platform. In this case we are considering JADE Leap module.

Before generating the code we can check whether the specification is complete, which means that it has enough information elements and consistency for the particular module that we intend to apply. This is done by the "verify" option in menu Modules-> Code Generation (see Figure 11).

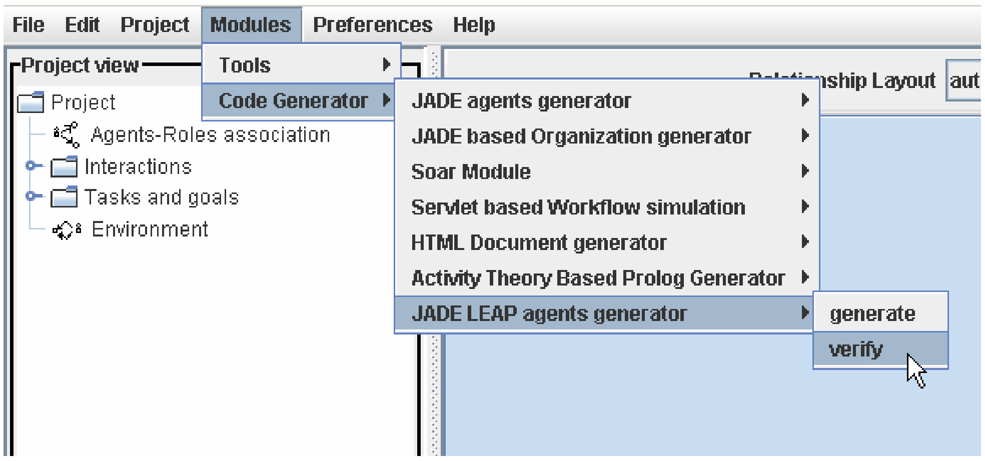
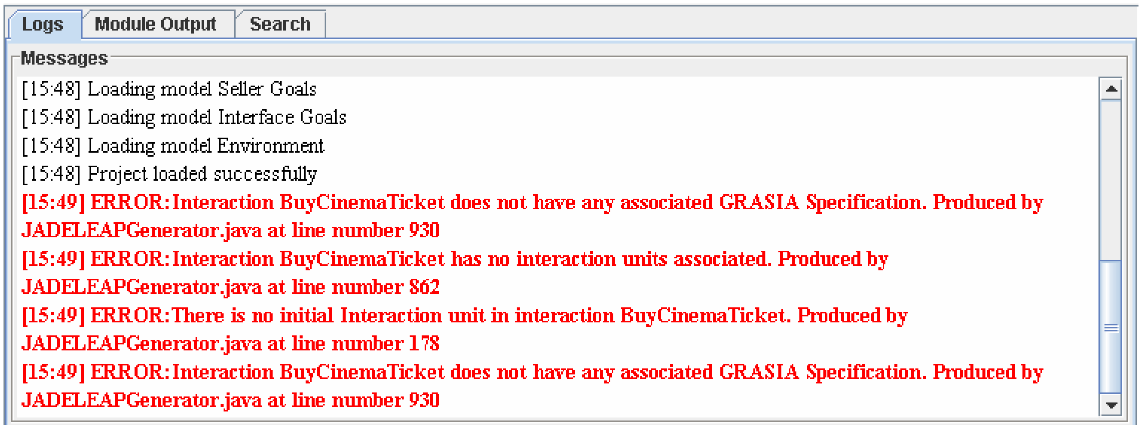


Figure 11. Invocation of a module

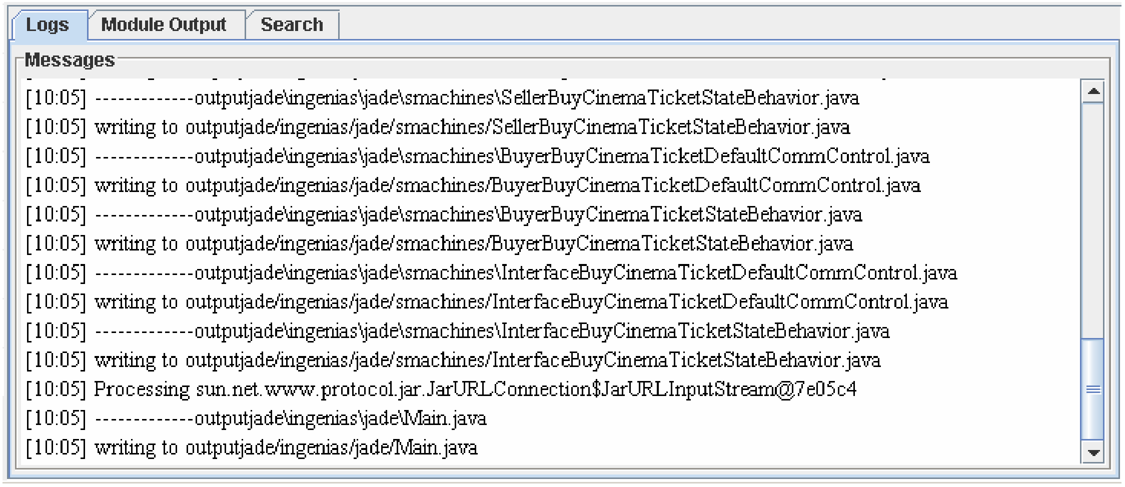
If something is wrong, the appropriate messages appear, as in Figure 12.



```
[15:48] Loading model Seller Goals
[15:48] Loading model Interface Goals
[15:48] Loading model Environment
[15:48] Project loaded successfully
[15:49] ERROR:Interaction BuyCinemaTicket does not have any associated GRASIA Specification. Produced by
JADELEAPGenerator.java at line number 930
[15:49] ERROR:Interaction BuyCinemaTicket has no interaction units associated. Produced by
JADELEAPGenerator.java at line number 862
[15:49] ERROR:There is no initial Interaction unit in interaction BuyCinemaTicket. Produced by
JADELEAPGenerator.java at line number 178
[15:49] ERROR:Interaction BuyCinemaTicket does not have any associated GRASIA Specification. Produced by
JADELEAPGenerator.java at line number 930
```

Figure 12. Errors in the verification of a specification

But if everything goes right, a celestial music sounds and a message “Specification is correct” appears in the Logs window. In this moment we can try the code generation, which is just using the option “generate” instead of “verify”. In fact, the generate option can be invoked without the verification, but this last is faster and useful to see what is missing in a specification.



```
[10:05] ------------outputjade\ingenias\jade\smachines\SellerBuyCinemaTicketStateBehavior.java
[10:05] writing to outputjade/ingenias/jade/smachines/SellerBuyCinemaTicketStateBehavior.java
[10:05] ------------outputjade\ingenias\jade\smachines\BuyerBuyCinemaTicketDefaultCommControl.java
[10:05] writing to outputjade/ingenias/jade/smachines/BuyerBuyCinemaTicketDefaultCommControl.java
[10:05] ------------outputjade\ingenias\jade\smachines\BuyerBuyCinemaTicketStateBehavior.java
[10:05] writing to outputjade/ingenias/jade/smachines/BuyerBuyCinemaTicketStateBehavior.java
[10:05] ------------outputjade\ingenias\jade\smachines\InterfaceBuyCinemaTicketDefaultCommControl.java
[10:05] writing to outputjade/ingenias/jade/smachines/InterfaceBuyCinemaTicketDefaultCommControl.java
[10:05] ------------outputjade\ingenias\jade\smachines\InterfaceBuyCinemaTicketStateBehavior.java
[10:05] writing to outputjade/ingenias/jade/smachines/InterfaceBuyCinemaTicketStateBehavior.java
[10:05] Processing sun.net.www.protocol.jar.JarURLConnection$JarURLInputStream@7e05c4
[10:05] ------------outputjade\ingenias\jade\Main.java
[10:05] writing to outputjade/ingenias/jade/Main.java
```

Figure 13. Logs tab when generating code

When running the generate option, the Logs window tab shows the files that have been created (Figure 13). If there are no error messages, we can go to the compilation and execution of the output files in order to see how agents work on a real platform, Jade in this case.

2.4. RUNNING THE AGENTS

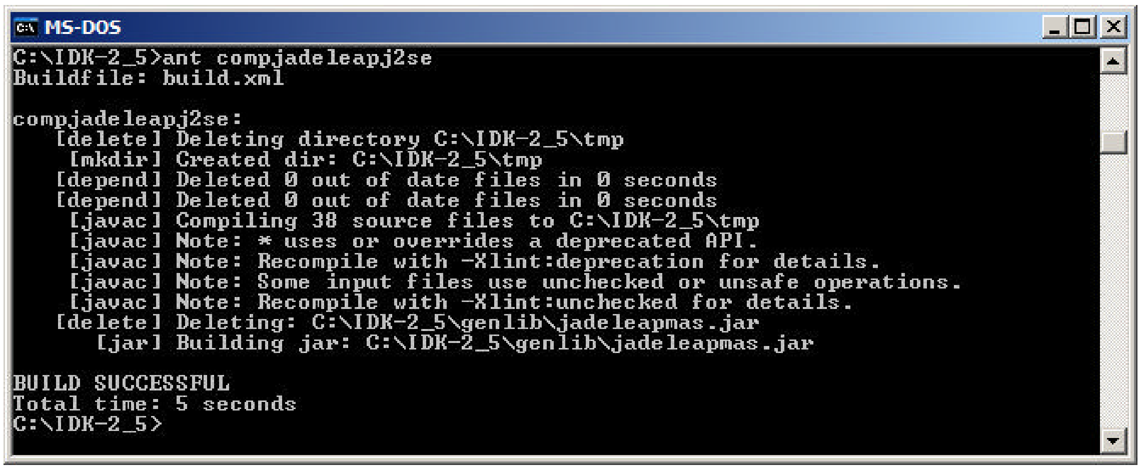
In order to facilitate the process, the IDK distribution provides some facilities to compile and execute the generated agents on the Jade platform.

Open a command or shell window and move to the directory of the IDK distribution.

There, execute

```
ant compjadeleapj2se
```

and you get something as in Figure 14.



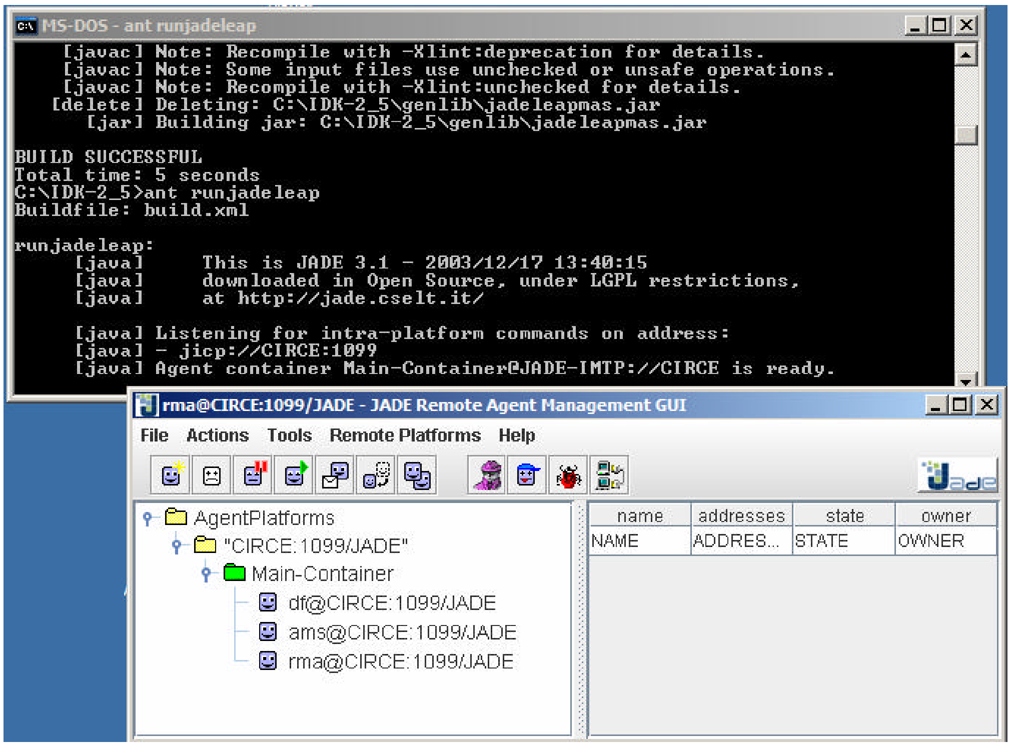
```
C:\IDK-2_5>ant compjadeleapj2se
Buildfile: build.xml

compjadeleapj2se:
   [delete] Deleting directory C:\IDK-2_5\tmp
    [mkdir] Created dir: C:\IDK-2_5\tmp
   [depend] Deleted 0 out of date files in 0 seconds
   [depend] Deleted 0 out of date files in 0 seconds
    [javac] Compiling 38 source files to C:\IDK-2_5\tmp
    [javac] Note: * uses or overrides a deprecated API.
    [javac] Note: Recompile with -Xlint:deprecation for details.
    [javac] Note: Some input files use unchecked or unsafe operations.
    [javac] Note: Recompile with -Xlint:unchecked for details.
   [delete] Deleting: C:\IDK-2_5\genlib\jadeleapmas.jar
      [jar] Building jar: C:\IDK-2_5\genlib\jadeleapmas.jar

BUILD SUCCESSFUL
Total time: 5 seconds
C:\IDK-2_5>
```

Figure 14. Compiling for Jade Leap the generated agents code

To run the system, in one console type:

```
ant runjadeleap
```

and the JADE platform opens as in Figure 15. This platform offers several containers for agents. Initially, it has the agents that provide basic FIPA services, such as directory facilitator and agent management system.



```
    [javac] Note: Recompile with -Xlint:deprecation for details.
    [javac] Note: Some input files use unchecked or unsafe operations.
    [javac] Note: Recompile with -Xlint:unchecked for details.
   [delete] Deleting: C:\IDK-2_5\genlib\jadeleapmas.jar
      [jar] Building jar: C:\IDK-2_5\genlib\jadeleapmas.jar

BUILD SUCCESSFUL
Total time: 5 seconds
C:\IDK-2_5>ant runjadeleap
Buildfile: build.xml

runjadeleap:
     [java]     This is JADE 3.1 - 2003/12/17 13:40:15
     [java]     downloaded in Open Source, under LGPL restrictions,
     [java]     at http://jade.cselt.it/

     [java] Listening for intra-platform commands on address:
     [java] - jicp://CIRCE:1099
     [java] Agent container Main-Container@JADE-IMTP://CIRCE is ready.
```

name	addresses	state	owner
NAME	ADDRES...	STATE	OWNER

Figure 15. JADE agent management

To run the agents, just type in another console:

```
ant runjadeleapmas
```

and the agents will execute.

Each agent has one window to interface for this experiment, as the one of Figure 16 for the SellerAgent. As all the windows start in the same coordinates, you should move them to other places in the screen to visualize all.

Initially the three agents are in the Container-1 and to start the experiment we can first put this agent in the Main-container by pressing the button Move on the right.

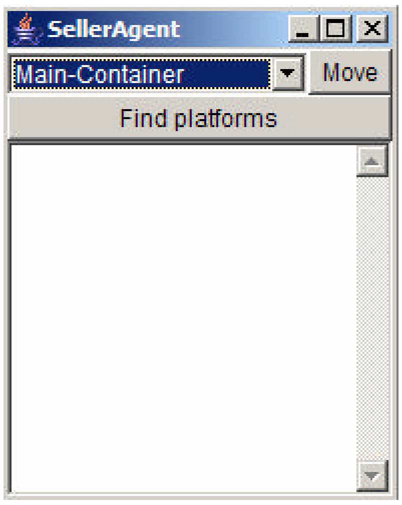


Figure 16. Interface of the SellerAgent

To start the application, the user has to request to the InterfaceAgent to buy the cinema ticket. This can be done by pressing the lower button on the corresponding window (see Figure 17). When doing so, this agent will request to the BuyerAgent to get the ticket, the BuyerAgent will travel to the Main-Container to find the SellerAgent, negotiate to get the ticket, and come back to Container-1, the original location, to provide this to the InterfaceAgent.

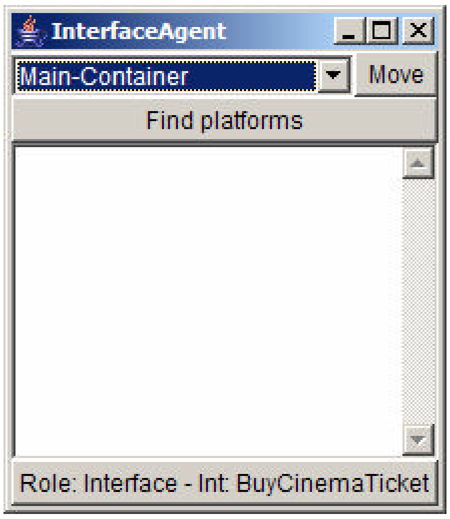
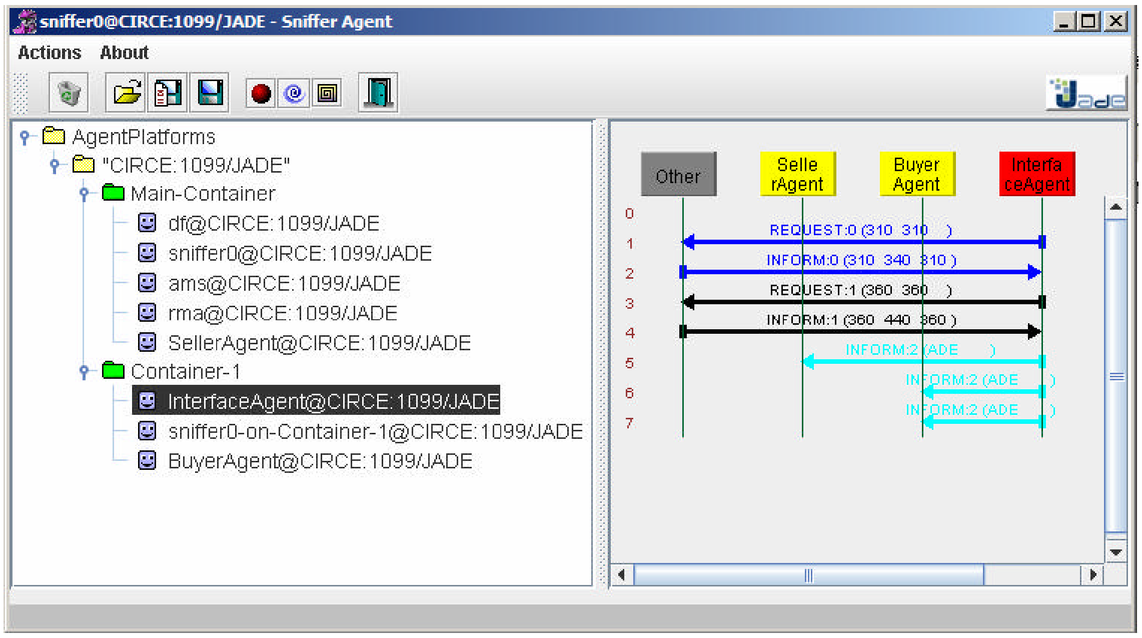


Figure 17. Interface of the InterfaceAgent

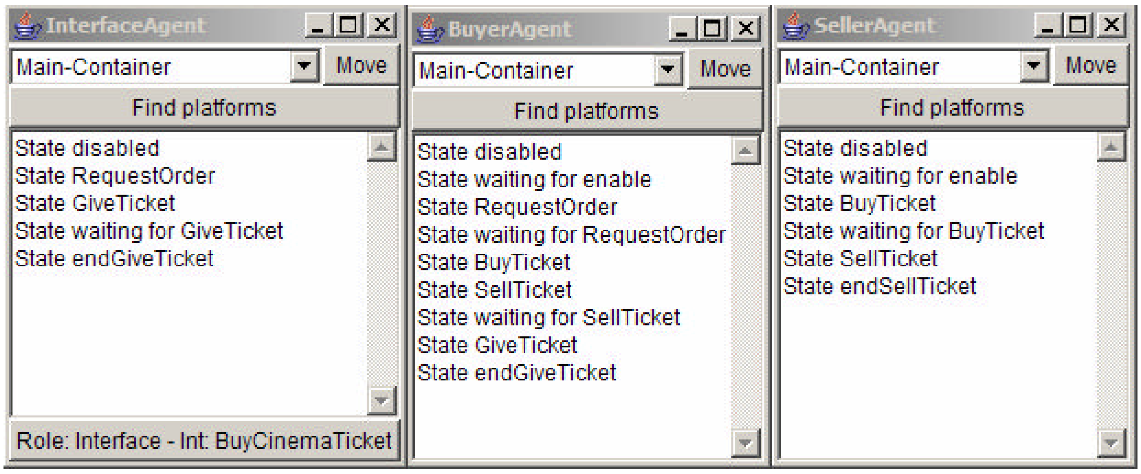
This process can be monitored in the JADE platform by observing the agents at each container and with the Sniffer Agent to see the different interactions, as shown in Figure 18.



AgentPlatforms
"CIRCE:1099/JADE"
Main-Container
df@CIRCE:1099/JADE
sniffer0@CIRCE:1099/JADE
ams@CIRCE:1099/JADE
rma@CIRCE:1099/JADE
SellerAgent@CIRCE:1099/JADE
Container-1
InterfaceAgent@CIRCE:1099/JADE
sniffer0-on-Container-1@CIRCE:1099/JADE
BuyerAgent@CIRCE:1099/JADE

Figure 18. Snapshot of the interactions captured by the JADE Sniffer Agent

The state of each of the agents is shown in their respective windows. At the end they look like in Figure 19.



InterfaceAgent:

State disabled
State RequestOrder
State GiveTicket
State waiting for GiveTicket
State endGiveTicket

Role: Interface - Int: BuyCinemaTicket

BuyerAgent:

State disabled
State waiting for enable
State RequestOrder
State waiting for RequestOrder
State BuyTicket
State SellTicket
State waiting for SellTicket
State GiveTicket
State endGiveTicket

SellerAgent:

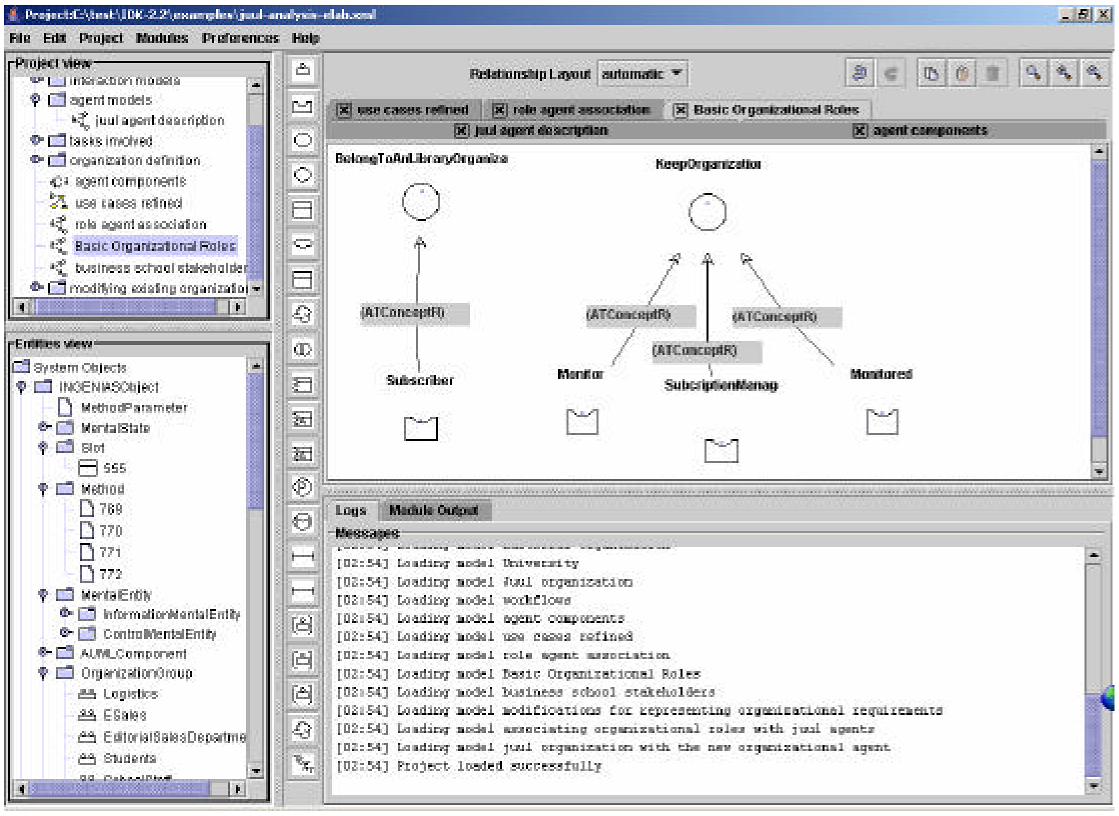
State disabled
State waiting for enable
State BuyTicket
State waiting for BuyTicket
State SellTicket
State endSellTicket

Figure 19. States that have passed each agent in the experiment

3. THE GRAPHICAL EDITOR

The main purpose of the graphical editor (Figure 20) is to create and modify the specifications of multi-agent systems (MAS). A specification here is a set of diagrams that represent different views of a MAS. The diagrams constitute a project, and they are organized using package-like constructs.

The editor saves these specifications using XML, so that other external tools can analyse them and produce other kind of outputs. Also, the editor provides access in runtime for installed plugins, and is able to load new plugins in run-time. These plugins are called in this document *modules* (which are described in section 5).



```
[02:54] Loading model University
[02:54] Loading model Juul organization
[02:54] Loading model workflows
[02:54] Loading model agent components
[02:54] Loading model use cases refined
[02:54] Loading model role agent association
[02:54] Loading model Basic Organizational Roles
[02:54] Loading model business school stakeholders
[02:54] Loading model modifications for representing organizational requirements
[02:54] Loading model associating organizational roles with juul agents
[02:54] Loading model juul organization with the new organizational agent
[02:54] Project loaded successfully
```

Figure 20. A screenshot of the editor

3.1. PARTS OF THE EDITOR AND RELATED OPERATIONS

Figure 21 shows the different parts of the editor. Each part is described in the following sections.

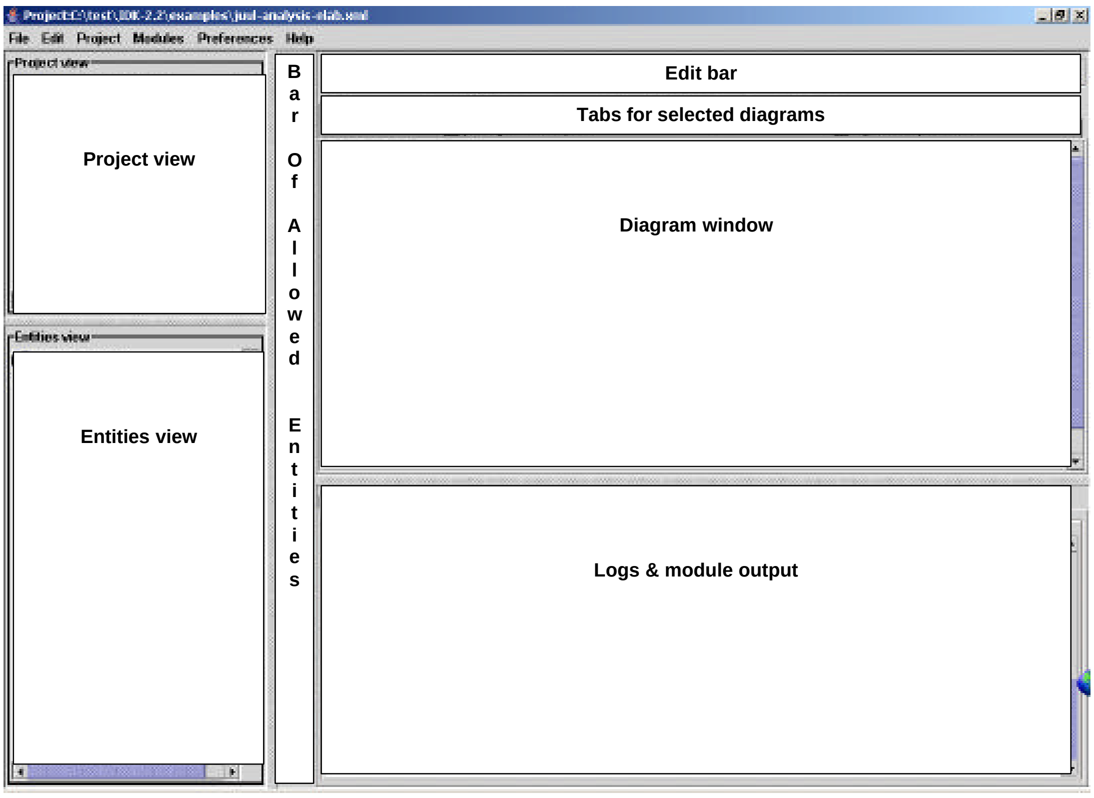

Project:C:\test\IDK-2.2\examples\jul-analysis-elab.xml
File Edit Project Modules Preferences Help
Project view
Project view
Entities view
Entities view
Bar Of Allowed Entities
Edit bar
Tabs for selected diagrams
Diagram window
Logs & module output

Figure 21. Parts of the editor

3.1.1 Project view

The Project view presents, organized as a tree, the different diagrams of a project. Diagrams are represented with special icons, and are the leaves of the tree. Diagrams can be grouped in packages, which are represented as folders. Frequent actions in the Project view are:

- **Drag and drop**. You can drag and drop a diagram into a package or a package into another package.
- **Open a diagram**. By double clicking the left mouse button on the diagram name or icon, the diagram will open in the diagram window.
- **Create a package**. To create a package, first select a package, then click the right button of the mouse and select *add package* in the pop up menu. Once selected, write down a name.
- **Create a diagram**. To create a diagram, first select a package, then click the right button of the mouse and select the type of diagram in the pop up menu. Once selected, write down a name.
- **Remove**. Select the diagram or package, and then click the right button of the mouse and select *remove package/diagram.*
- **Rename**. Select the diagram, and then click the right button of the mouse and select *rename*. Write down the new name, and then accept.
- **Modify properties of a diagram**. Some diagrams have special properties. These can be accessed through the pop-up menu triggered on right clicking on a diagram icon. Properties are modified the same way as object properties.

3.1.2 Entities view

The *Entities view* contains a tree-like view of the types of entities that exist in the specification. The tree shows types as well as current type instances (the entities in the specification, which may appear in several diagrams). Types are represented by folders. Type instances are distinguished by icons different of folders.

Several operations can be performed on each entity by selecting the corresponding icon (not a folder) and pressing the right button of the mouse. Then a pop-up menu appears that shows several operations:

- **Add the selected entity to the current diagram**. It creates a copy of the entity in the diagram, but only if the diagram can handle that specific type. An entity can appear in several diagrams.
- **Remove the entity**. It removes the entity from all diagrams and set all attributes of entities pointing at it to null.
- **Edit properties**. It shows different properties associated to the selected entity.

3.1.3 Diagram editor

The Diagram editor of IDK consists of three parts:

- The *Edit Bar*, with common operations for the edition (undo, redo, zooming, copy/paste, etc.)
- The *Bar of allowed entities*, which is specific for each type of diagram. It has buttons that allow to create instances of particular entity types in the current diagram. Normally, all entity types that are allowed for a particular diagram appear in the bar.

  In order to know what type of entity is represented by a button, just move the mouse over a button and its name will appear.

  In order to insert a new entity in the current diagram, press one of the buttons. The new entity will be allocated in the top-left of the *diagram* window.
- *Diagram windows*, which are organized as tabbed windows, one for each diagram that has been opened.

  Selected diagrams are presented in several tabs in the diagram window. For each open diagram there is a tab that is labelled with its title (tabs for selected diagrams). The label of the current diagram is highlighted in light grey. The labels of the other diagrams appear in dark grey.

Diagrams can be managed as follows:

- **Open a diagram.** Select a diagram in the Project view and click twice with the mouse right button. A new tab will appear and the diagram will be selected as current diagram.
- **Close diagram.** Click with the mouse in the cross at the left of the tab header, and the diagram will be closed (the tab is suppressed).
- **Select as current diagram.** Clicking on the tab of one diagram makes it appear in the main window to view and modify it.

Frequent actions with the current diagram are:

- **Insert a new entity.** When right clicking in the diagram window, a pop-up menu will appear with the different valid entities that can be included in the diagram. This action is the same as pressing an entity button in the *allowed entities* bar.
- **Connect two entities with a relationship**. There are two ways to connect entities in a diagram.

  Figure 22 shows how to connect an *agent* and a *goal*. First, put the mouse on the little square in the middle of one of these figures. Then drag from there to the other entity. The target entity highlights when the relationship has found the destination. In that moment, release the mouse button. A new window will appear, showing different possible valid relationships that could be defined (in case that the relationship is not allowed in this diagram the new window will notice this). You have to select one (also when there is only one possible relationship type you have to acknowledge it). Afterwards, it asks how to configure the extremes of the relationship, since, sometimes, several assignments are valid (although normally those selected by default are the best option). On finishing, a new relationship is created.

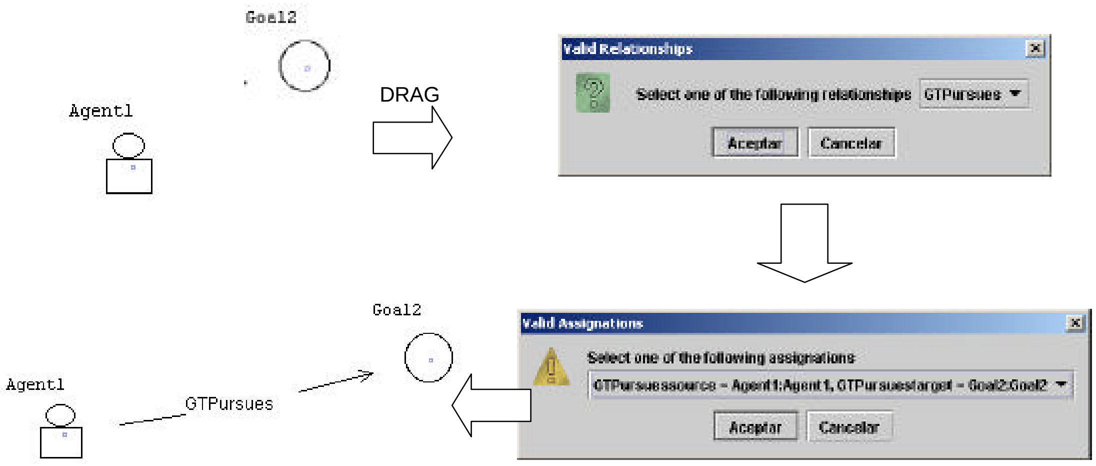

Goal2
Agent1
DRAG
Valid Relationships
Select one of the following relationships
GTPursues
Aceptar
Cancelar
Goal2
Agent1
GTPursues
Valid Assignations
Select one of the following assignations
GTPursuessource - Agent1:Agent1, GTPursuestarget - Goal2:Goal2
Aceptar
Cancelar

Figure 22. Steps to create a relationship among two entities

- **Adding a new entity to an existing relationship**. Some relationships accept more than two entities, i.e., they are n-ary. When you have already created a relationship and want to add another entity, the process is simple. First, you move the mouse to the relationship to which you want to connect the entity until your see that the mouse icon changes. Then, drag the mouse towards the entity that you want to connect and release the mouse. Following, a window will appear showing what kind of role will play the new entity in the relationship. Select one and accept the new type. If the entity cannot be accepted by the relationship, because of its type or the cardinality of the relationship, an error dialog will appear.

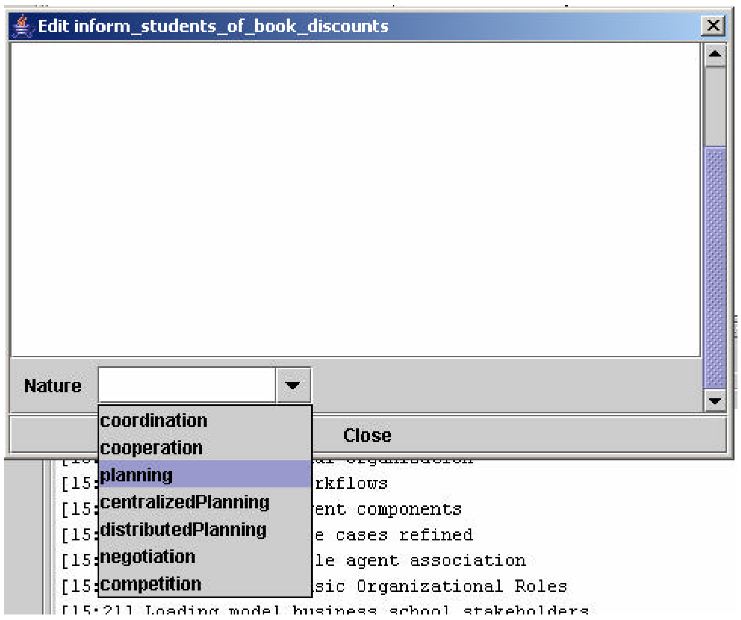
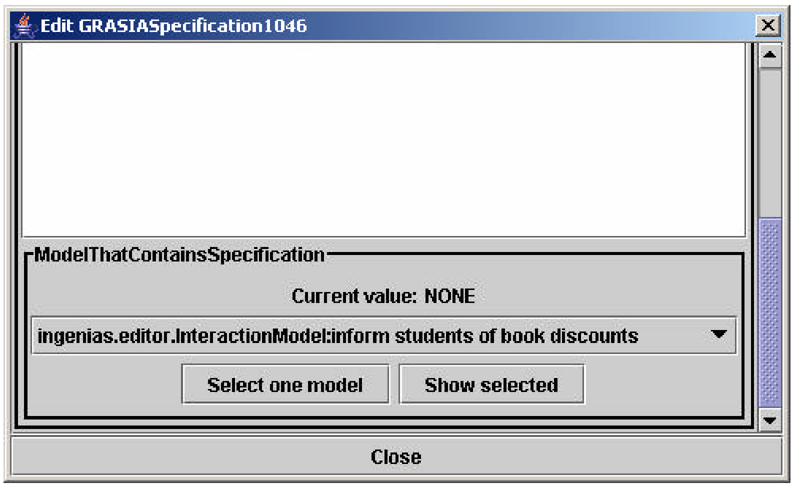


Figure 23. A window for editing an entity, showing a combo box with fixed values

- **Edit an entity**. By double left-clicking on an entity. A new window will appear with data that can be edited in several ways:

- **Text fields.** Just write whatever you want to. It should admit the ISO-8859-1 character set.
- **Combo box fields.** These fields can admit only values defined in the associated list (see Figure 23). A value has to be selected.
- **Diagram reference fields.** This field allows to refer to other diagrams (see Figure 24). The procedure consists in selecting in the combo the name of the diagram. The combo will show only existing diagrams of a preconfigured type. Once selected, press *Select one model*. This will make the *current value* label change. To jump to the selected diagram, press *show selected*.


Edit GRASIASpecification1046

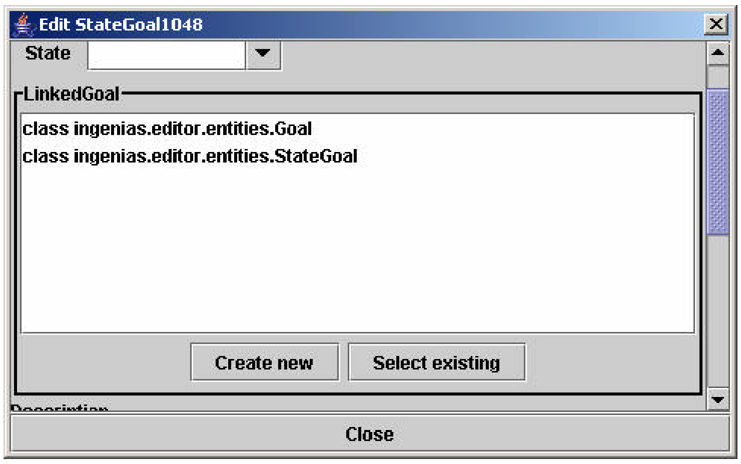
ModelThatContainsSpecification

Current value: NONE

ingenias.editor.InteractionModel:inform students of book discounts

Figure 24. An editing window for an entity showing a field that refers to another diagram

- **List box fields**. These fields are used to store references to entities already defined in some diagram or create new entities. They also can refer to a collection of values or a single value.
  - **Collection.** The list should appear initially in blank. By left-clicking in the list, a pop-up menu will appear with four options:
    - **Add existing.** A dialog window will appear with a combo box showing valid already defined entities that could be used. Select one and press *yes.*
    - **Add new.** A dialog window will appear with a combo box showing valid types of entities that could be used. Select one and press *yes.* Another window will appear to fill in the data of the new entity.
    - **Open selected.** It opens a window that shows the data of the selected entity. This window allows the same functionality to edit the data as presented here. So proceed recursively.
    - **Remove selected.** It removes the entity from the list but not from the main repository visible in the *Entities View.*
  - **Single value.** The list (see Figure 25) will show possible types that could be allowed in that part of the definition. You have to select one of them and press one of the available buttons. With *Create new*, you will create a new instance of the selected type and associate it directly with this field. The new instance will also appear in the *Entities View.* With *Select Existing*, you will associate this field with an existing entity. Existing entities of selected type will be shown in a dialog window in form of a combo box.


Edit StateGoal1048

State

LinkedGoal

class ingenias.editor.entities.Goal

class ingenias.editor.entities.StateGoal

Figure 25. An editing window for an entity showing how to modify single value entity field.

- **Changing the icon of an entity.** Some entities have different associated views. You can select one of them by right clicking on an element and going to the *views* option. There, available views will be shown (see Figure 26).

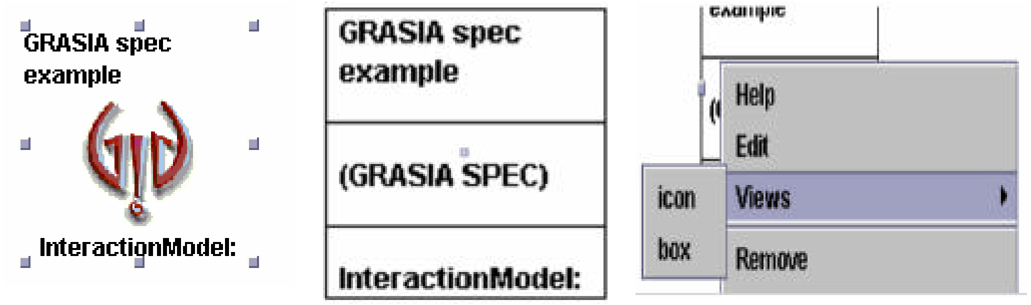


Figure 26. Different views associated with a GRASIA Specification. First view is the *icon* view, and the second the *box* view

- *Edit bar.* This bar offers options to
  - **Zoom/unzoom**. By pressing the and the buttons. The zoom will return to its normal state when pressing the button.
  - **Redo/undo actions**. By pressing the and buttons. Undo/redo actions should be limited to changes of positions of diagram components. It will not work to undelete entities, relationships, or unedit changes made to properties.
  - **Copy/paste/delete**. By pressing the , , or respectively. Relationships cannot be copied or pasted. If some are selected, the editor will unselect them automatically. Deleting an entity requires to delete first the relationships it participates into or the edges that connect the to-be-deleted entity to the relationship. In some cases, when the mandatory arity of the relationship would be violated, there is no other option but deleting the relationship before.
  - **Relationship layout**. It sets how relationships are lay out into the diagram. *Automatic* stands for *allocating the relationship in the middle of all participating entities*. *Manual* stands for *you are responsible for allocating the entity*. By default, Automatic layout.

3.1.4 Logs & Module output.

This window shows messages from the editor and from different loaded modules.

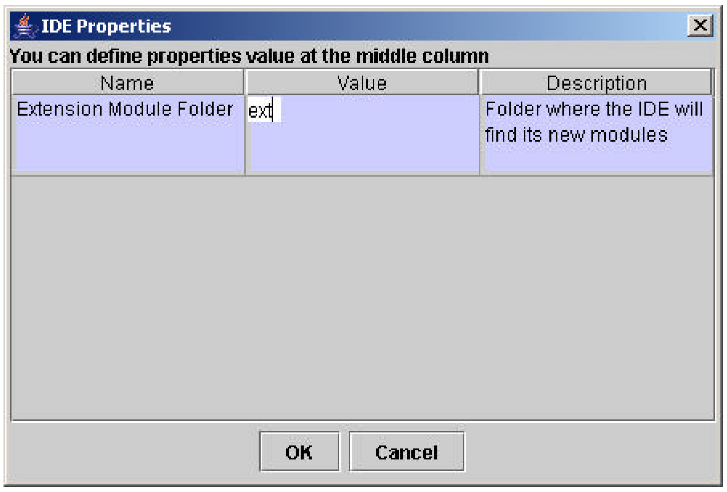
The window can be cleared by right-clicking in it. A pop-up menu will appear with a *clear* option. Select it.

3.1.5 Main menu

The Main menu provides access to some key functionality.

- **File menu**. This menu contains a list of recently loaded/saved files. By clicking on one of them, it will be loaded. Also, you have the usual options here (load,save, save as) that do not deserve further explanation. Besides, the *new* option creates a new empty project. This can be useful to start from scratch.
- **Edit menu**. It has some of the functionalities of the *edit bar* (copy,paste,delete,undo,redo). Also, it includes three more:
    - ***Select all***. This option selects all entities in a diagram. It is useful to move all entities in the diagram. It is not good for deleting, since it may cause error dialog windows to appear.
    - ***Copy diagram to clipboard.*** It creates an image of the current diagram and stores it into the system clipboard.
    - ***Copy diagram to file.*** It creates an image of the current diagram and stores it into a file with two different formats: *jpeg* and *png*. We advise to use png, since it ensures the diagram quality.
- ***Project menu***. It permits to add diagrams or packages to a selected package exactly as it can be done in the Project view. Select the kind of diagram or the package and write down its name in the dialog window that will be shown.

  Also, this menu provides access to the *project properties window*. This window, see Figure 27, shows a table with three columns. The first is the name of the property, the second its value, and the third some text that describes the purpose of the property. Initially there is only one property, the one that defines where to look at new plugins or modules. However, whenever a new module is loaded, it can define new properties that will be added to this window. Usually, this property refers to configurable execution parameters of the module. Note that you can only modify the second column, the one titled *values*.



You can define properties value at the middle column

Name	Value	Description
Extension Module Folder	ext	Folder where the IDE will find its new modules

Figure 27. Properties of a project. You can edit only the *value* column of the window.

- **Modules menu.** This menu allows the execution of modules installed in the editor. Each module has an entry that can be allocated in the *tools* or *code generator* section (see Figure 28). The *tools* entry contains modules whose main purpose is not generating code but analizing the specification to generate reports or detect inconsistencies, for instance. The *code generator* entry contains modules that generate code from diagrams. The concrete procedure will be explained later in this document.

By now, it is enough to know that modules can both generate code and verify properties of a set of diagrams. The list of modules can be updated if a developer allocates a new module in the extension folder. Also, if the module has the same name as an existing one, the new version will replace the old one.

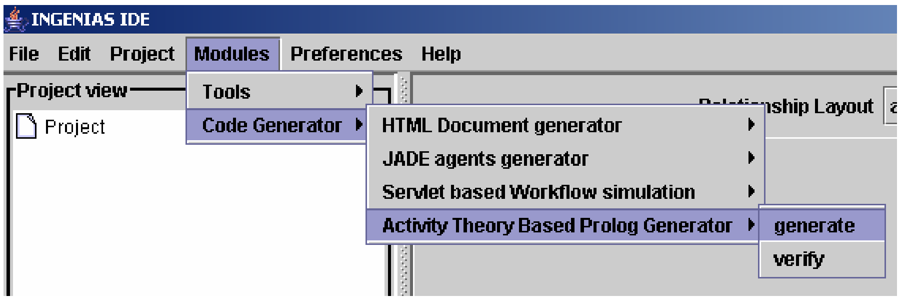


Figure 28. Module list available in the 2.2 version of the IDK

- **Help menu**. It provides access to a summarised version of this document (*tool manual* option), the credits, and the possibility of forcing a garbage collection to optimise memory usage (*Force GC* option). Please note that according to SUN specifications, it seems that calls to the garbage collector do not imply an immediate garbage collection.

3.2. WORKING WITH THE EDITOR

Using the editor requires some knowledge about the INGENIAS methodology, specially for the notation. There are several types of diagrams with several types of entities in each one. For a quick guide we recommend one of the papers mentioned in section 1.1 or chapter 7. Some examples are given also in chapter 6.

In general, the use of the editor is quite simple, but here are some advices

1. **Think in advance what diagrams you will need and create packages for them in the project.** Packages are very useful to organize the different diagrams, specially for the specification of complex systems. For instance, if you need diagrams to describe how an agent performs a task A, create a folder with a label something like *specification of task A execution*(see Figure 29)

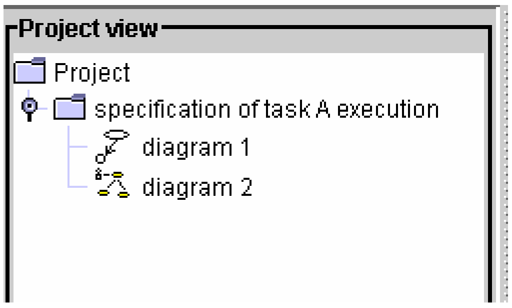
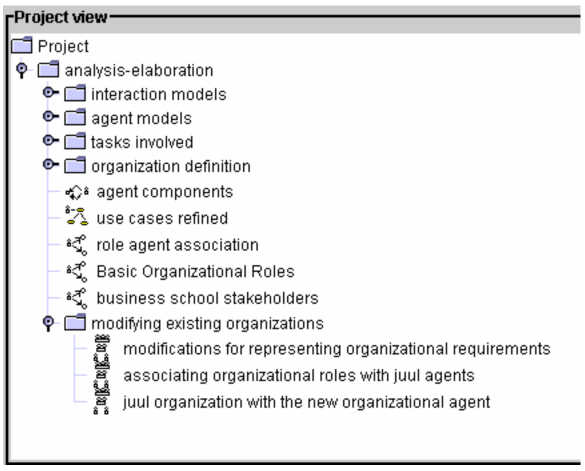


Figure 29. Organize your work with packages

2. **Use meaningful names.** This will help to trace diagrams and make the documentation more readable. Also, fill in the description fields of each entity and diagram when possible.
3. **Do not be afraid if there are too many diagrams.** This is usual in any conventional development, why not in an agent oriented one? That is why we recommend to start using packages from the beginning. Anyway, you can always create them later and use the drag&drop feature to rearrange them.



Project view

Project

- analysis-elaboration
  - interaction models
  - agent models
  - tasks involved
  - organization definition
  - agent components
  - use cases refined
  - role agent association
  - Basic Organizational Roles
  - business school stakeholders
  - modifying existing organizations
    - modifications for representing organizational requirements
    - associating organizational roles with juul agents
    - juul organization with the new organizational agent

Figure 30. The number of diagrams can grow easily if you get into details. See how a package structure helps to manage them

4. **Copy diagrams to the clipboard.** Remember that the editor allows you to copy a diagram to your favourite text editor by copy&paste. You can also save the image to a file for later use.

4. AUML

The IDK Editor includes an alpha version of AUML Protocol diagrams. These diagrams are defined according to the last AUML draft (October 2004) that can be found at http://www.auml.org.

4.1. Creating a protocol diagram

AUML Protocol diagrams are created as other types of diagrams, by clicking on the *project* folder in the *Project view* (see Figure 31) or the *Project* option in the main menu. In any of these two cases, you have to select *AUML Interaction Diagram*

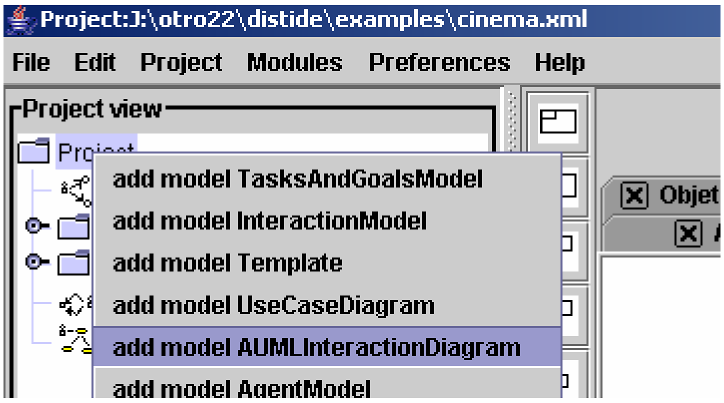

Project:J:\otro22\distide\examples\cinema.xml
File
Edit
Project
Modules
Preferences
Help
Project view
Project
add model TasksAndGoalsModel
add model InteractionModel
add model Template
add model UseCaseDiagram
add model AUMLInteractionDiagram
add model AgentModel
Objet

Figure 31. Selecting an AUML Protocol diagram type

The icon that represents a diagram of this type is the same as the one belonging to a common interaction (⛙).

4.2. Defining protocols

The protocol definition starts by pressing the *protocol* button in the diagram button bar. This corresponds to the first button as shown in Figure 32.

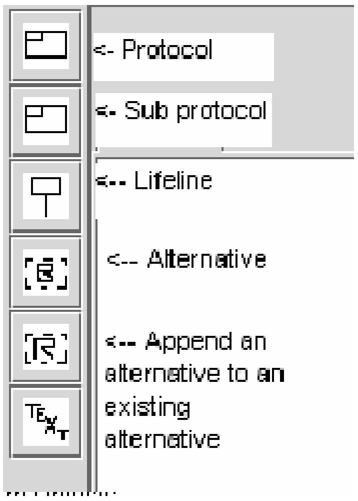
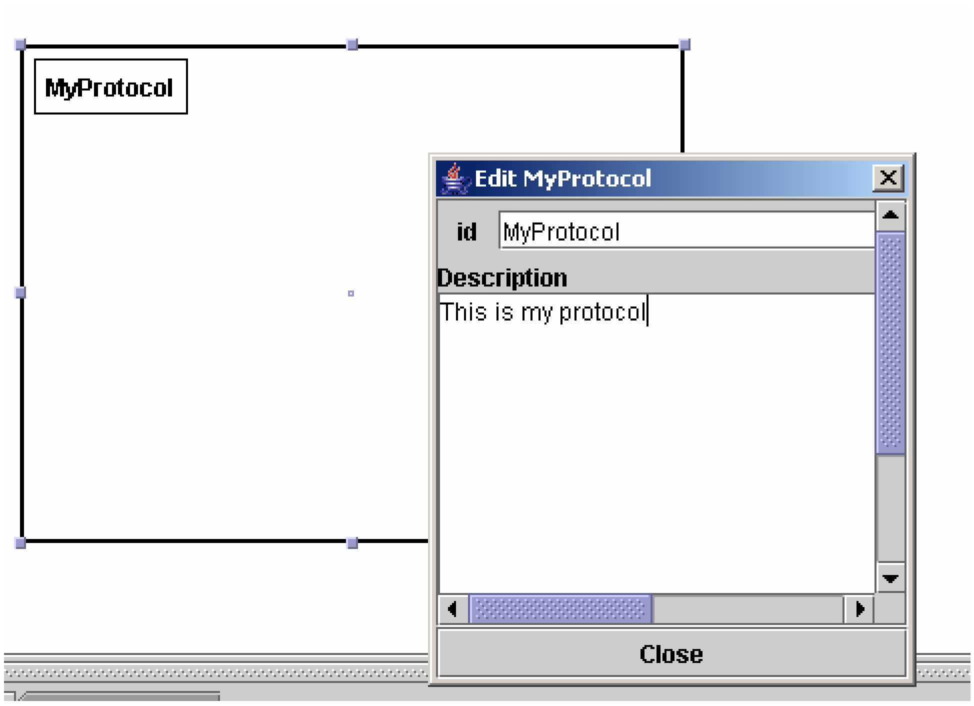


Figure 32. Buttons for defining the agent protocol

As a result, a protocol box is created. Initially, this Box only shows the title of the protocol (see Figure 33). You can edit the name of the protocol by modifying the id field, and this will be changed in the diagram.



id: MyProtocol

Description: This is my protocol

Figure 33. Protocol box renamed to *My protocol*

Once the protocol box is created, it is time to create some life lines. In this implementation, a lifeline is an entity that represents an role or an agent instance that is going to send messages to other roles or agents.

To create a lifeline, a protocol box has to be selected first. Lifelines are created with one of the buttons that you can see in Figure 32. The result can be seen in Figure 34.

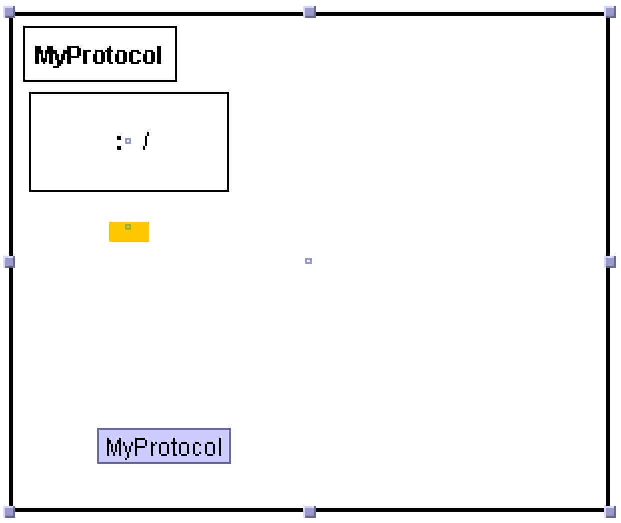
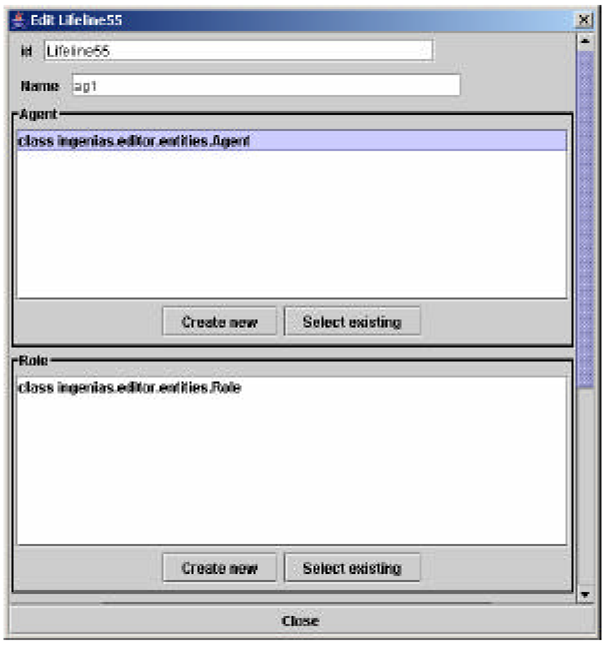


Figure 34. Protocol box with a lifeline

Lifelines do not show any special information until they are edited. Relevant data is (see Figure 35):

- **Name:** The name of the instance.
- **Agent:** the type of agent that represents this instance. You can create a new kind of agent or select an existing type.

- **Role:** The role that the agent instance is playing at the moment. You can create a new kind of role or select an existing one.


Edit Lifeline55

id Lifeline55

Name ag1

Agent

class ingenias.editor.entities.Agent

Role

class ingenias.editor.entities.Role

Figure 35. Data that personalizes a life line

After modifying the different fields, you will see that the representation of the lifeline incorporates the new data (see Figure 36).

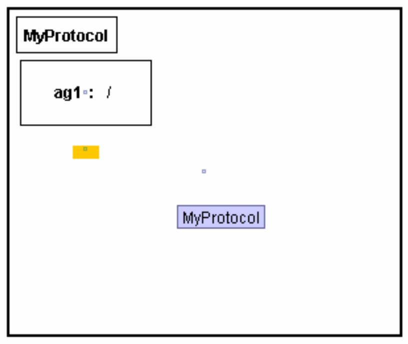


Figure 36. Modified lifeline after typing the name of the agent instance

Now, we will create another lifeline and start defining the messages among agents. The process is the same as before, but this time we previously moved the first lifeline to the right so that there is enough space for the new one. Please, remember to select the protocol box before creating the lifeline.

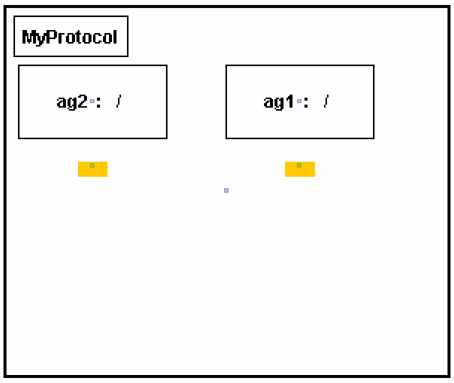


Figure 37. Two lifelines in a protocol box

4.3. Connecting two lifelines

Connecting two lifelines is the first step for defining a message passing, an alternative definition, or a protocol instantiation. To connect, notice the little square drawn in the orange rectangle. By dragging from one to the equivalent on the other lifeline, we get something like the following.

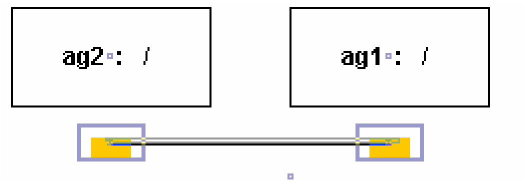


Figure 38. A connection between two lifelines

The process is the same as when connecting two entities as shown in section 3.2. So the next step is selecting the kind of association between the two lifelines. At the moment there are three.

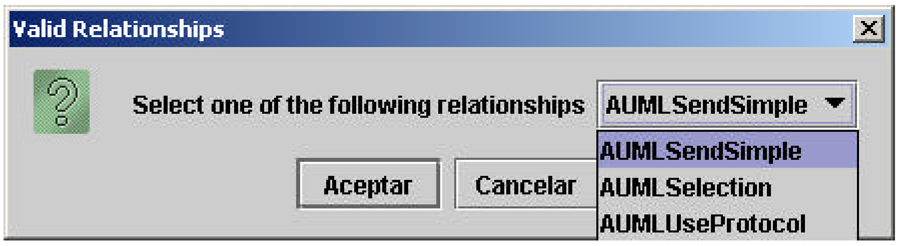


Figure 39. Dialog window for determining the type of connection between two lifelines

What happens when selecting each of these options is explained in the following sections.

4.4. Sending a simple message

Sending a message is represented as shown in Figure 40, with an arrow from one lifeline to another and a square in the middle that will hold message data. Notice that after creating the arrow, the orange squares are longer.

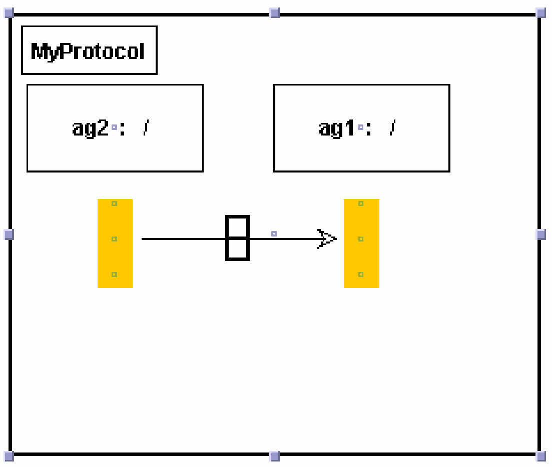
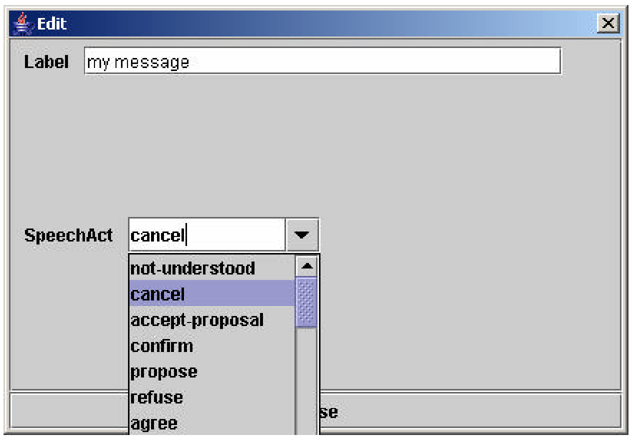


Figure 40. Result of selecting "AUML Send simple"

The data that can be configured in the message is:

- **Label.** This label will be shown in the diagram in the top part of the message.
- **Speech act.** It is a combo with predefined names of valid FIPA ACL performatives (speech acts).



Label: my message

SpeechAct: cancel

not-understood
cancel
accept-proposal
confirm
propose
refuse
agree

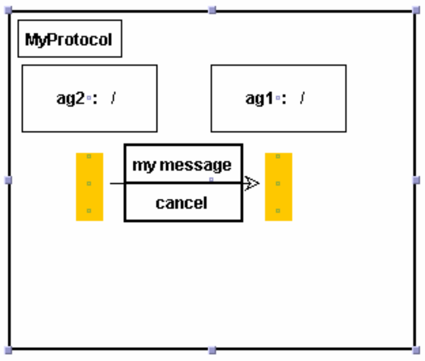


Figure 41. Data for a message (left) and resulting drawing (right)

You can repeat the process as many times as you need.

4.5. Creating an alternative

You create an alternative when you select AUMLSelection in Figure 39. As a result, the editor creates an alternative box between the two lifelines (see Figure 42).

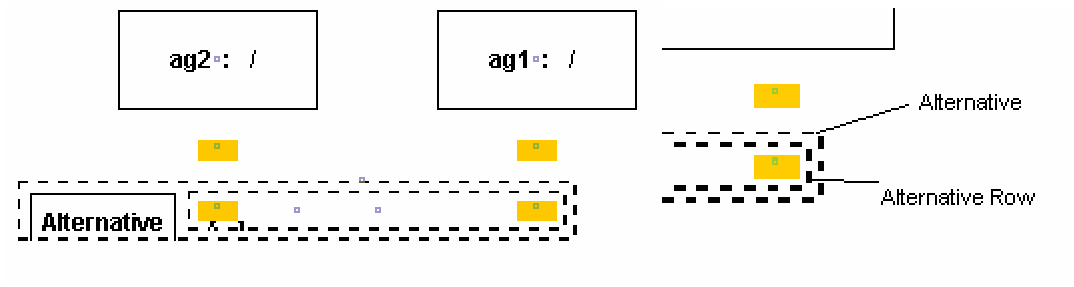
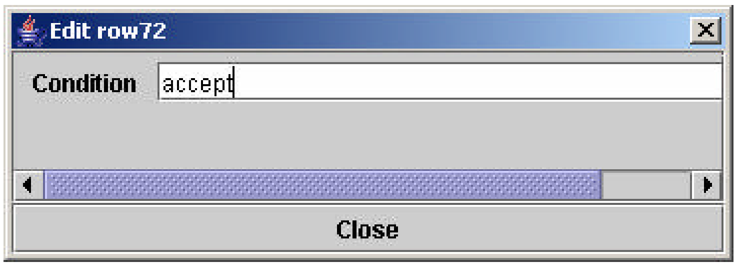


Figure 42. Alternative between ag1 and ag2

Figure 42 shows an alternative and an alternative row. The first one is just a container for the different possibilities that may consider a message selection in the middle of a protocol. Each one of these possibilities is an alternative row. Alternative rows can be edited and their attributes modified. In this case, the attribute is the condition that this alternative row represents.



Edit row72

Condition: accept

Figure 43. Defining a condition for an alternative row

You can draw messages from one alternative lifeline to another, just as you did in the previous section (see Figure 44)

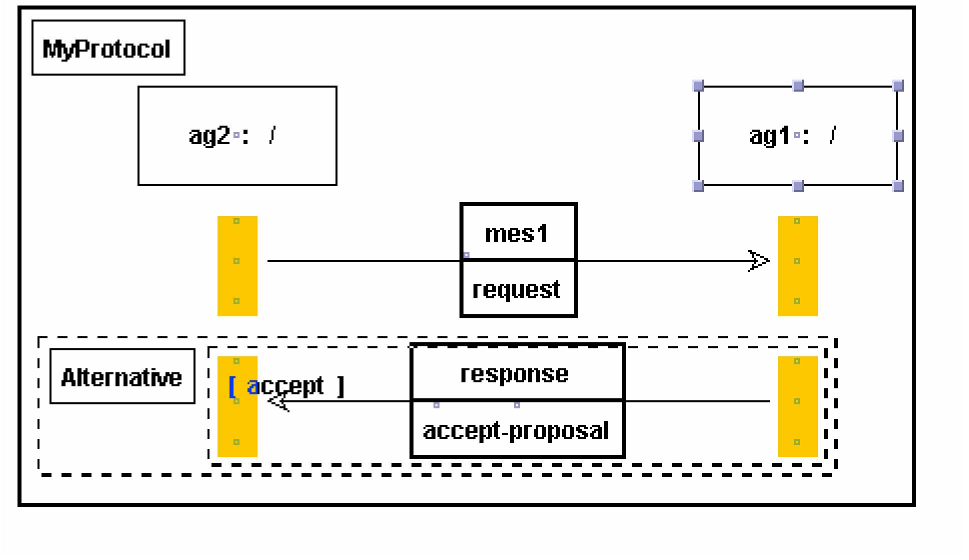


Figure 44. Drawing messages between two lifelines with an existing alternative

To add a new Alternative Row, we select the alternative again and press the alternative row button.

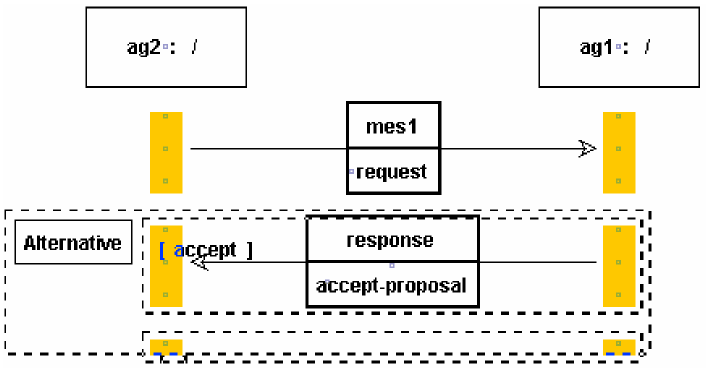


Figure 45. Defining another alternative

Again, we define a condition for the alternative row and define the message as well. The result is shown in Figure 46.

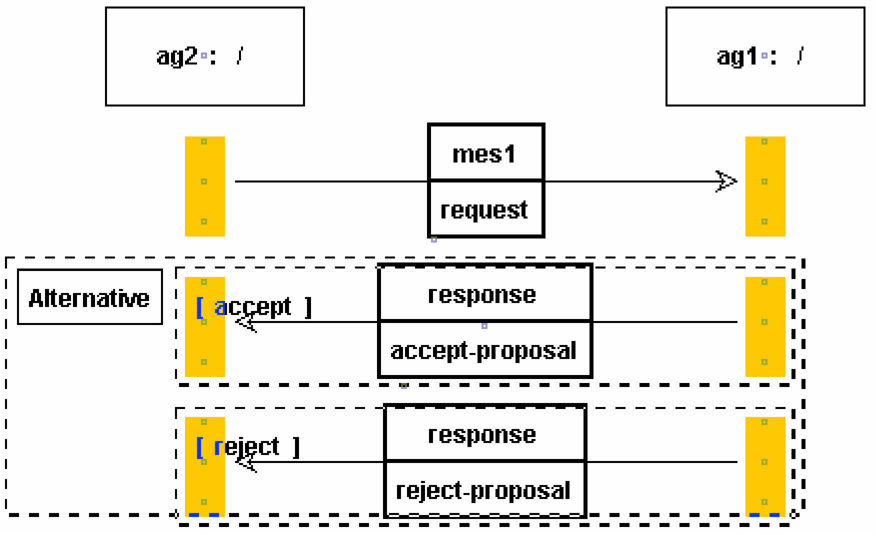


Figure 46. Alternative with two possible responses from ag1 to a request sent by ag2

4.6. CREATING A SUB PROTOCOL

A sub-protocol refers to another protocol in order to make use of it. In the example of Figure 47 we create another protocol in a different diagram and intend to reuse a previously defined diagram, in this case, the one defined in Figure 46. We start the definition of the subprotocol as in Figure 39 and select *AUMLUseProtocol*.

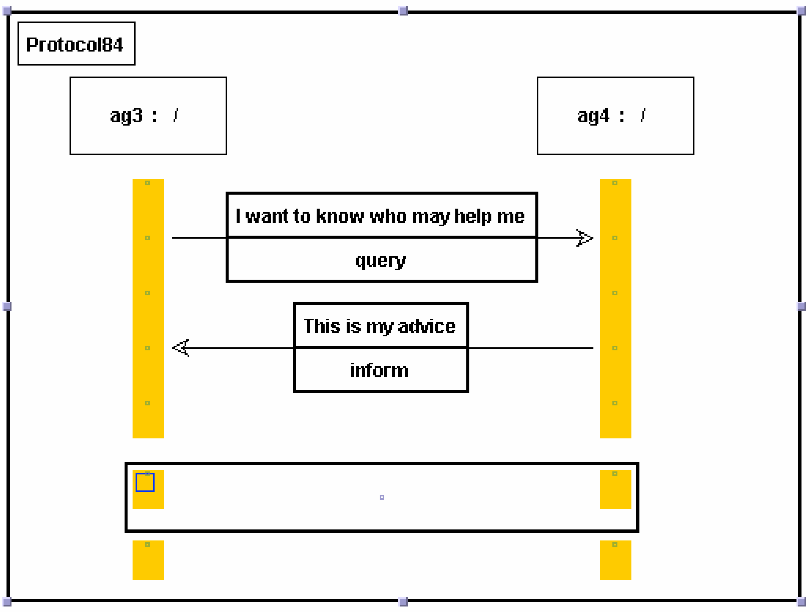


Figure 47. Subprotocol creation

Figure 47 shows a subprotocol. It is a rectangle that needs to be instantiated. To instantiate it, we select it and try to edit its containment (right click → select edit or double click).

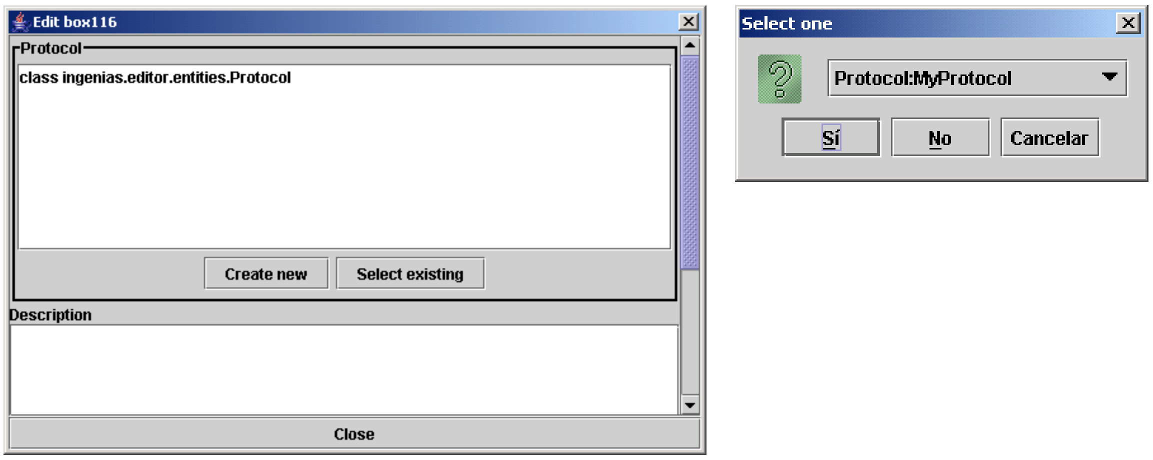


Figure 48. Edit dialog to describe the subprotocol properties

Figure 48 shows the edit dialog. There, we have to choose either *create new* or *select existing*. In this case, as we want to reuse an existing protocol, we chose *select existing* and then *My protocol*, which is the one we defined in Figure 46. The result is shown in Figure 49.

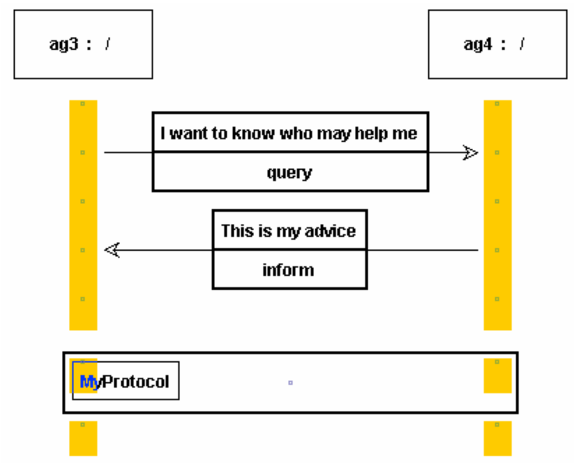


Figure 49. Initialized subprotocol instance

4.7. TO BE DONE

There are several aspects that have not been completed in this version (that's why it is an alpha). In concrete, we could highlight:

- **Layout problems**. Alternatives and Subprotocols are always appended at the bottom of the life lines. This is correct for alternatives, but not for subprotocols. Besides, the gap between orange squares is something we want to remove.
- **Implement other AUML protocol diagrams primitives.** So far, only alternatives, basic messages, and subprotocols have been implemented.
- **Semantics of AUML primitives.** Throughout the implementation of the diagrams, many questions have arosen that were not considered in the FIPA AUML Draft diagrams. For instance, what kinds of connections were allowed. Is it correct to nest a subprotocol in an alternative? Does the lifeline end in an alternative (can we define messages after an alternative)? Is it possible to add another lifeline to an existing alternative if the new lifeline has already defined an alternative? These an other questions make us think that there are still missing aspects in AUML that need to be further detailed.

5. MODULES

Modules (see Figure 50) are programs that process specifications and produce some output:

- **Source code.** There is an infrastructure that facilitates the generation of source code. The infrastructure bases on templates defined with XML. These templates are filled in with information extracted from the diagrams.
- **Reports.** Diagrams can also be analysed to check, for instance, whether they have certain properties or whether special semantics (defined by the developer) are being respected, or to collect statistics of usage of different elements.
- **Modifications on current diagrams.** Though this feature is in beta stage, a module could insert and/or modify entities in the diagrams, or insert/delete diagrams. This feature is useful to define personal assistants that interact with the tool.

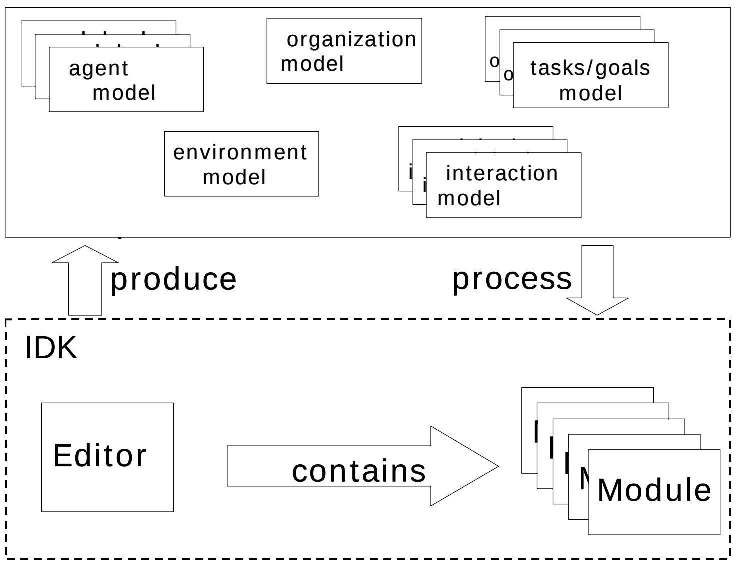


Figure 50. Relationship between the IDK, models and components

5.1. USE OF MODULES

The IDK already provides a collection of modules. New modules are integrated easily by leaving their code files in the extensions folder (to determine the extensions folder see how to configure IDK properties in chapter 2) and the IDK will automatically detect them.

Available modules are invoked from the IDK editor in the Main menu ->Modules. Each module may have particular usage instructions. In this chapter we describe at the end some of the modules already present in the IDK distribution, and how to use them.

5.2. DEVELOPMENT OF NEW MODULES

Modules are built in the top of a framework that provides facilities to traverse specifications, extract information from specifications, and put the extracted information into templates.

Developing a module for code generation is usually an iterative process through several steps. These steps are described by indicating the tools for performing them and the expected output for each one. The basis for code generation is the availability of code templates of he concrete target platform. The developer needs first to define the architecture of the code for the target platform, and then the transformation from specification to the code templates.

1. **Producing a prototype of the application.** Initially, a developer would centre into one or more features, easy to implement if possible.

   a. **Tools:** A conventional development environment (depending on the programming language and target platform, in the case of code generation).

   b. **Output:** A prototype of an application in the target platform that realizes a small part of the specification, with a selected set of features.

2. **Mark up the prototype code.** Parts of the prototype should match against parts of the specification. As a result, a developer identifies the possible mappings from the specification to the prototype code.

   a. **Tools:** An XML editor or a text editor.

   b. **Output:** Prototype code marked up with tags. The marked-up pieces of source code are known as *templates*.

3. **Generate/modify a module.** The module will traverse the specification and obtain the information required by the prototype. This can be done with a conventional Java development environment (IDK libraries are written in Java), including IDK module development packages (which are described in the following sections).

   a. **Tools:** Java development environment.

   b. **Output:** One or more Java classes that extend *BasicToolImp* or *BasicCodeGeneratorImp* classes. Other classes may be created as well.

4. **Deploy the module**. Java classes and templates are put together into a jar file. This jar file is put in a specific folder where the IDK Editor can load it.

   a. **Tools:** J2SDK and *ant* (http://ant.apache.org). The J2SDK jar tool generates the jar, and the *ant* tool executes the appropriate *ant* task to perform the compilation and copy of the source and binary files.

   b. Output: A *jar* file that has the module code and the templates obtained from the prototype.

5. **Test the module.** Testing tasks are launched from the IDK Editor. By executing the module over the specification, the developer can check if the diagram is traversed properly and if all templates have been filled in as they should. Also, as templates demand concrete information, it may be possible that this is not present or that it is not expressed as it should. Therefore, it may turn out that the specification was not correct or incomplete.

   a. **Tools:** The IDK editor mainly. Other tools may be needed when testing the output code (appropriate compilers and runtime environments).

   b. **Output:** Problems with the code generated by the module, problems with the traversal of the specification, or problems with the specification.

6. **Debug.** If something goes wrong, debug the prototype and go to:

   a. *Step 2.* If there is new code that was not marked up before.

   b. *Step 3.* If the failure was in the module and the data traversal.

   c. *Step 4.* If there was a failure in the prototype and could be solved without marking up the code again.

7. **Refinement and extension.** When the module is finished, it can translate diagram specifications into code or perform some verification of properties. However, the module performs these tasks with a reduced set of the diagram specification. The next step would be to take the code generated by the module and extend it so that it can satisfy other parts of the specification. Therefore, we would go back to step 1.

Modules produce code using a template based approach. As an example, Figure 51 shows how to generate code for a rule based system, in this case JESS (Friedman-Hill, 2002). A developer defines a template of a JESS rule and extracts data from the MAS specification to generate the rest of rules. Rules need a condition, an action, and a name. These data are expressed using a concrete structure that will be presented later. As a result, we get two different rules, which are instantiated from the same template.

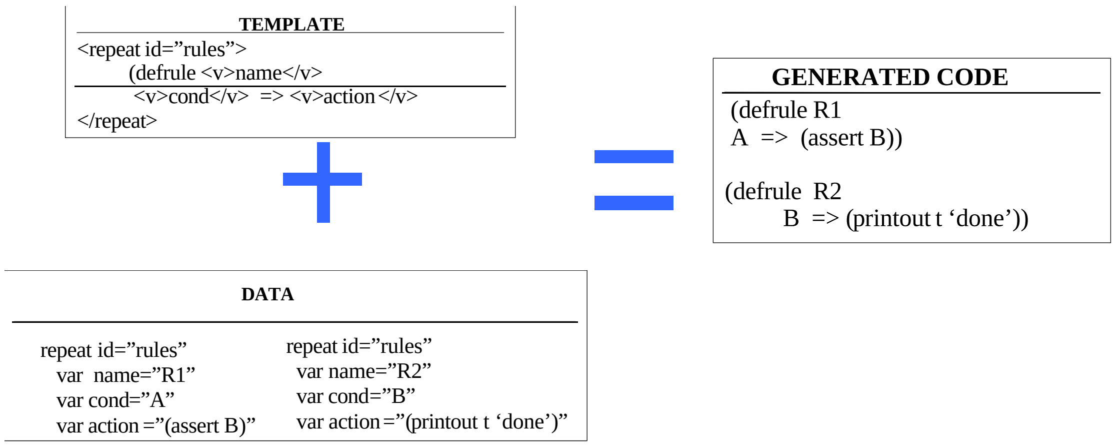


Figure 51. An example of code generation for a set of JESS rules

According to this description, the reader may infer that we assume that:

- There are parts of the specification that are very similar among themselves.
- There are parts of the code that are very similar among themselves.

One may be a consequence of the other, since code is supposed to satisfy a specification, and if there are parts of the specification that are repeated, there should be parts of the code that repeat as well.

A module can be of two different types: a code generator or a specification processor. To create a module of the first type, you have to extend the *ingenias.editor.extension. BasicCodeGeneratorImp* class. If you try to create a module of the second type (e.g., a verification tool), you have to extend the *ingenias.editor.extension.BasicToolImp* class.

Both classes define abstract methods that have to be redefined into their inheriting classes. Also, both classes initialise internal variables that give access to the internal data structure of the IDK. Once created the corresponding derived class, developers will realize that most of the work is done, and that only traversal specification and template creation needs to be done.

In the next sections, there are further instructions for some of these tasks. Section 5.2.1 describes how to traverse the specification graph. Section 5.2.2 presents how to create templates from prototypes. Section 5.2.3 introduces the facilities to create the data needed to fill in the templates. Finally, Section 5.2.4 explains how to deploy a module.

5.2.1 Traversing the specification

A module views the internal data of the IDK Editor as the data structure in Figure 52. This data structure is an interpretation of the GOPRR model (see section 7.2 for more details). In this representation, a *graph* is a diagram, a *relationship* is an edge from an object to another object, and an *entity* is an object.

A relationship here is n-ary. This means that a relationship may have many ends, not only a source or a target. This capability is useful for representing agent concepts relationships, since there are many of this kind.

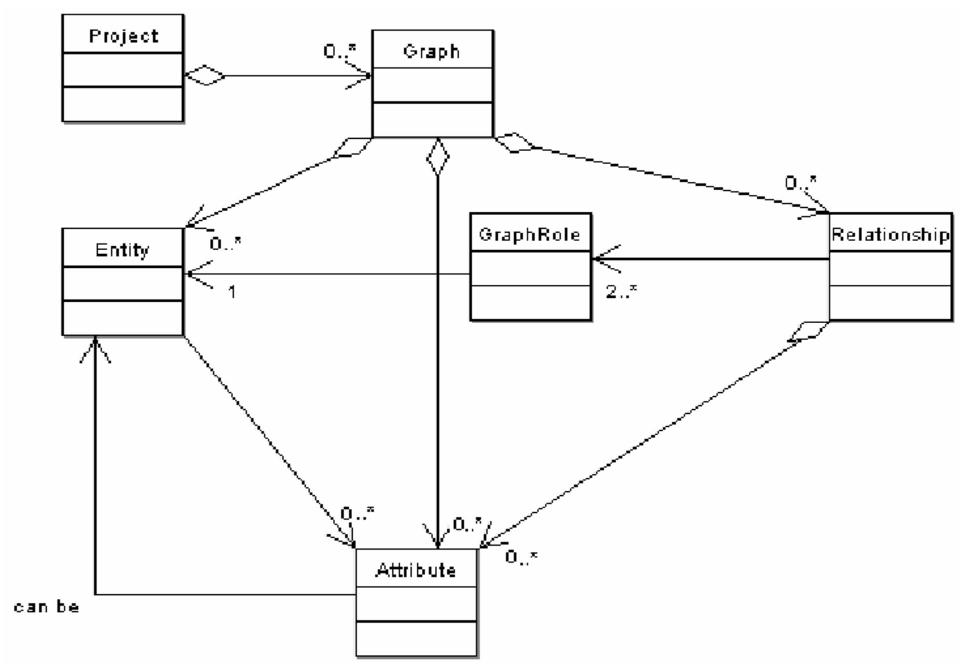


Figure 52. Logical view of the data stored in the IDK

The access to this structure is controlled by a set of interfaces, which are shown in Figure 53. These interfaces take advantage of the commonalities of the elements of Figure 52, such as that all elements have properties (graphs, relationships, and entities).

In the IDK, a developer can obtain instances of *Graph*, *GraphRelationship*, and *GraphEntity* by using a Singleton pattern (Gamma, Helm, Johnson, & Vlissides 1995). A class that implements this pattern in the IDK is the *ingenias.generator.browser.BrowserImp* and the method to invoke to get a valid instance of this class is *getInstance*( ):

```
Browser browser=BrowserImp.getInstance();
```

This kind of instantiation is valid only when the module is executed inside the IDK Editor. If you plan to execute outside the editor directly over an specification file generated by the IDK Editor, you should put into your main method the following:

```
File file;
....
ingenias.editor.Log.initInstance(new java.io.PrintWriter(System.err));
ingenias.generator.browser.BrowserImp.initialise(file);
Browser browser=BrowserImp.getInstance();
```

Traversing the specification means to define a graph traversal algorithm that goes through elements of the specification and:

1. Ensure that all requested elements are present. A traversal intends to find certain elements and, from them, go to other elements in the diagrams.
2. Extract information from the requested elements. Information extraction is a matter of invoking specific methods of *GraphEntity*, *Graph*, and *GraphRelationship*.

Initially, the developer has a list of existing graphs or a list of existing entities in all graphs. At this point, it is important to clarify that some objects may be already present in different diagrams. From these initial graphs or entities list, the developer articulates the traversal. It can be as simple as "*In each diagram, look for instances of the relationship X, and tell me what elements they connect*" or as complex as a traversal with the purpose of generating code for the Jade platform.

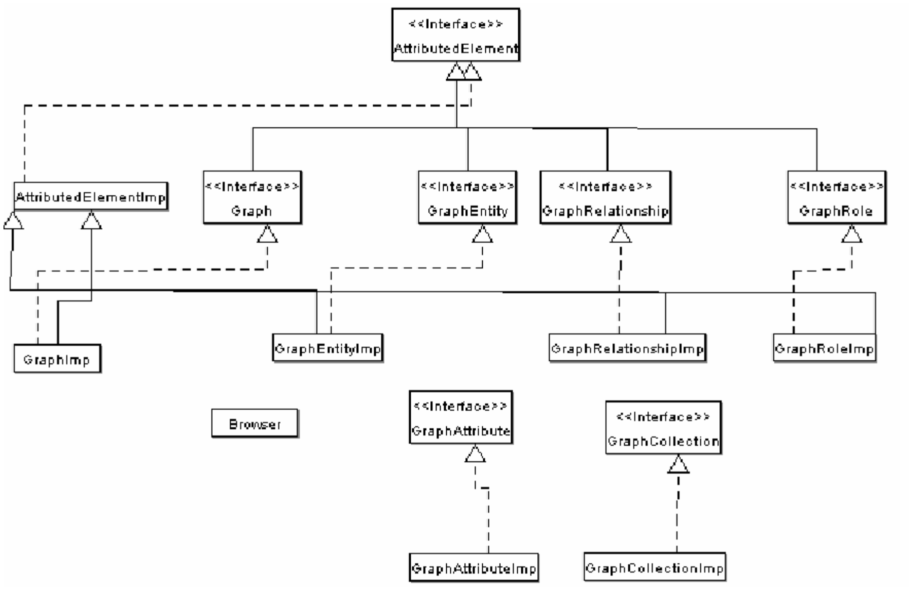


Figure 53. Interfaces provided to access the data stored as XML

A simple example of how to traverse existing diagrams and printing out their names is the following:

```
// browser has been previously initialised
Graph[] gs = browser.getGraphs();
StringBuffer result = new StringBuffer();
for (int k = 0; k < gs.length; k++) {
Graph g = gs[k];
result = result.append( "\n###### Diagram " + g.getName()+
                       " ######\n");
result.append(this.generaInformeDiagrama(g)+"\n");
}
System.out.println(result);
```

We use StringBuffer because the concatenation of Strings is inefficient and may lead to memory exhausted errors.

5.2.2 Marking up the prototype

The code of the prototype is marked-up according to the DTD shown in Figure 54. This DTD determines that any piece of source code is a XML document. Therefore, templates can be written in any language, provided that the source code can be later marked-up.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT file (#PCDATA | v)*>
<!ATTLIST file  overwrite (yes|no) #REQUIRED>
<!ELEMENT program (#PCDATA|repeat|saveto|v)*>
<!ELEMENT repeat (#PCDATA | saveto | v | repeat)*>
<!ATTLIST repeat id CDATA #REQUIRED>
<!ELEMENT saveto (file, text)>
<!ELEMENT text (#PCDATA | repeat | v | saveto)*>
<!ELEMENT v (#PCDATA)>
```

Figure 54. DTD for extracted information

Tags from Figure 54 have concrete semantics:

- **program**. It is the main tag of the document. It requires no special semantics.
- **repeat**. It means that the text among *repeat* tags has to be copied and pasted again to have a duplicate. The duplicate is parsed following looking for variable instantiation or other meaningful tags.
- **v**. It represents a variable. Its matching tag encloses a piece of text that has to be replaced. The text itself is considered as an id. This id permits to distinguish what data corresponds to this variable.
- **saveto.** This tag orders to save the contained text into a file. The file name and the text are enclosed into specific tags
    - o **file**. It is the name of the file. It can contain other tags as well.
    - o **text.** It is the text to be saved. It can contain other tags as well.

When writing templates, soon it becomes clear that is not easy to code programs as XML since < and > symbols, which appear frequently, have to be codified as &lt; and &gt; (as demands XML). Doing this for every symbol is a time consuming task. To save effort, we tend to express XML tags in our templates using the at symbol @, instead of < and >. This way, instead of writing:

```
<program>
 if (a &lt; b)
 cout &lt;&lt; "hello"
</program>
```

We would write:

```
@program@
 if (a < b)
 cout << "hello"
@/program@
```

Of course, other uses of @ symbol would be forbidden in the code. Both formats can be used to generate code, though specific methods should be invoked in each case.

So that this decision does not affect the rest of the framework, we have prepared some tools that translate one format to another. The command that performs the translation is the following:

```
java –cp "lib\ingeniaseditor.jar" ingenias.generator.util.Conversor [-a2t|-t2a] my_template
```

The a2t means transforming an @ format to the conventional XML format. The t2a is the opposite. This utility also transforms <, >, &, ‘, “ symbols to their equivalents in XML.

5.2.3 Generating the code

The code generation facilities take as input a template that satisfies the DTD from Figure 54 and data to fill the template *v* and *repeat* tags. The data that feeds the code generator has the structure shown in Figure 55. We name this data *sequences* due to some data structures we used in the past for this purpose. Right now, there are Java classes that implement this structure and provide adequate translation mechanisms for XML.

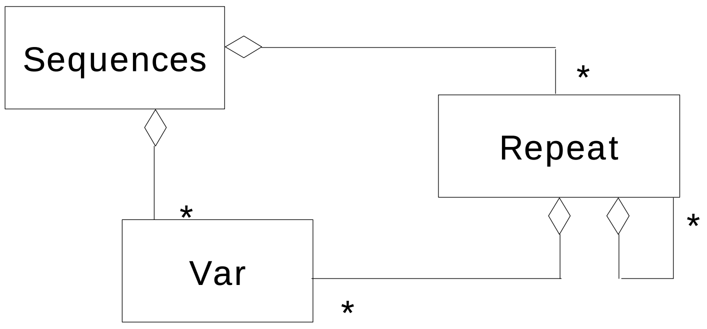


Figure 55. UML Description of the data structure

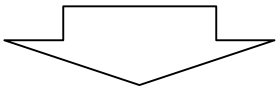
As Figure 55 remarks, there can be several *repeat* instances and *v* instances. Each one is created using an id, in the case of the *repeat*, and an id and data, in the case of the *var*. The id of *repeat* and *var* is used to distinguish among the different *v* and *repeat* tags that may exist along the template. Data is supplied as an unordered sequence of *repeat* or *var* structures and the effect is a replacement of the template by concrete data, in the case of *var* structures, or duplicates of existing data, in the case of *repeat* structures.

But, what data should be extracted and what structure should it have? We answer partially this question with an utility that parses a template and returns text representing how the data structure should look like. The utility is started from command line in the install folder of the IDK:

```
java
–cp "lib\ingeniaseditor.jar; lib\xerces_2_3_0.jar; lib\xercesImpl.jar; lib\xerces-
J_1.4.0.jar" ingenias.generator.util.ObtainInstantiationStructure
template_filename
```

As an example of the kind of output, Figure 56 shows the data structure needed to instantiate a template of the html code generator module included in the IDK distribution (this module produces HTML to document a MAS specification). The template corresponds to an *index.htm* file.

```
@program xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../plantilla.xsd"@
      @saveto@
            @file overwrite="yes"@
                  @v@output@/v@ /index.html @/file@
            @text@
<HTML>
<BODY bgcolor="#FFFFFF">

<p><img src="../images/logograsia.jpg" width="151" height="71"> </p>
<p><font size="5">Specification Diagrams are:</font>
  <BR>
</p>
<ul>

 @repeat id="paquete"@
       <li><font size="4"><b> @v@name@/v@ </b></font> </li>
       <ul>
       @repeat id="graph"@
            <li>Diagram name: <A HREF=" @v@name@/v@ .html">@v@name@/v@</A> type :
<font color="#000099">  @v@tipo@/v@  </font>
            </ li>

       @/repeat@
       </ul>
       <br>
  @/repeat@

<p><font size="3">Document generated automatically with the Ingenias Development
Kit <font color="#993333">
  IDK 2.1</font></font></p>

</BODY>
 </HTML>

@/text@
      @/saveto@
@/program@
```



```
v output
  repeat  id = package
    v name
   repeat  id =  graph
     v name
     v name
     v type
```

Figure 56. Information structure extracted from the template

A similar Java structure with the classes from Figure 55 would look like the following:

```
Sequences seq=new Sequences();
Repeat r1=new Repeat(“package1”);
Repeat r2=new Repeat(“package2”);
seq.add( r1 ); seq.add( r2 );
r1.addVar(new Var(“name”,”mipackage1”));
r1.addVar(new Var(“type”,”agent diagram”));
r2.addVar(new Var(“name”,” mipackage 2”));
r1.addVar(new Var(“type”,”interaction diagram”));
.....
```

And to launch the code generator, the following code should be executed. The input stream is a stream whose source is a file containing the template. The sequence structure is transformed into a string, whose *toString()* method is overloaded to generate the XML structure.

```
Sequence seq;
InputStream is;
...
ingenias.generator.interpreter.Codegen.applyArroba(seq.toString(),is);
```

The interpreter will analyse the template and will produce the output code. If there are *saveto* tags into the templates, the interpreter will save the results to the specified files. If not, the output will be the standard one (in the IDK's Logs and Modules window, see section 3.1.4).

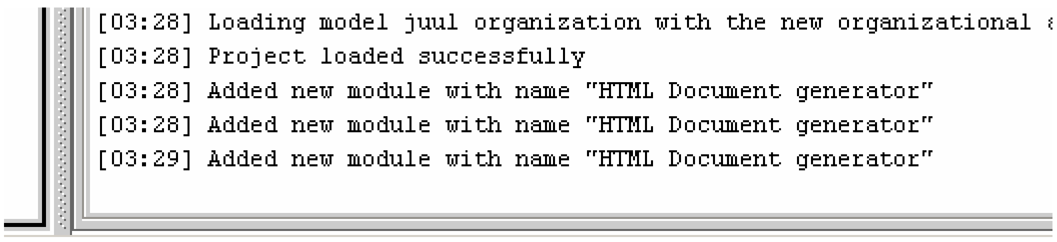
5.2.4 Deploying a module

A module has templates (in the case of code generation) and classes that extend the *BasicToolImp* or the *BasicCodeGeneratorImp*. All of them are placed in a folder created by the developer.

To deploy the module, templates must be allocated in a folder named *templates* in the root of the folder structure where module sources are. The deployment consists of:

1. Compiling the sources of the module into a separate folder, which we will call *binary folder*.
2. Copying the template folder into the binary folder.
3. Invoking the *jar* utility to compact the module binaries and the templates.
4. Moving the resulting *jar* to the deployment folder of the IDK. By default this is a folder named *ext,* which is located in the IDK install folder.

At the end of the process, if the IDK Editor is running, the message panel should show a message indicating that a new module has been added (see Figure 57). Each time you deploy a module the IDK Editor will load it automatically and replace internal references to it with the new version.



```
[03:28] Loading model juul organization with the new organizational
[03:28] Project loaded successfully
[03:28] Added new module with name "HTML Document generator"
[03:28] Added new module with name "HTML Document generator"
[03:29] Added new module with name "HTML Document generator"
```

Figure 57. Messages that confirm the module load

These tasks can be automatized if the developer uses the *ant* utility. In the *build.xml* file, the developer can find examples of how these tasks look like. For instance, the *modhtml* tasks, are specified as follows (bolder and italics represent comments inserted to facilitate understanding):

```
Change the location attribute to the path to your module source folder
 <property name="modhtmldoc" location="modules/srchtmldoc" />

      ....

 <target name="modhtmldoc">

 Here the folder structure is created
 <delete dir="${temp}" />

 <mkdir dir="${temp}/templates" />
 <depend srcdir="${modhtmldoc.dir}" destdir="${temp.dir}"

 cache="depcache">

 <include name="**/*.java" />

 </depend>
```

Now, module sources are compiled

```
 <javac compiler="modern" depend="true" destdir="${temp}"
debug="true">

 <src path="${modhtmldoc}" />

 <classpath>

 <pathelement path="${classpath}" />

 <pathelement path="${build}" />

 <fileset dir="lib">

 <include name="**/*.jar" />

 </fileset>

 </classpath>

 </javac>
```

After compilation, templates are copied to the binaries folder

```
 <copy todir="${temp}/templates">

 <fileset dir="${modhtmldoc}/templates"></fileset>

 </copy>
```

A jar is created with the name of the module

```
 <jar jarfile="${modhtmldoc}/modhtmldoc.jar"

 basedir="${temp}" />

 <delete file="${moddeploy}/modhtmldoc.jar" />
```

The resulting jar is moved to the deploy folder. Change the name of the module to avoid collision with other jars

```
 <move file="${modhtmldoc}/modhtmldoc.jar"

 toDir="${moddeploy}" />

 </target>
```

Only by changing the *modhtmldoc* property, you could get a personalized task to compile and deploy your module. The task would be started with:

```
ant modhtmldoc
```

Of course, we recommend not reusing completely the *ant* task code, copying and pasting the *modhtmldoc* tasks into your build.xml file, and modifying the copy trying to personalize if possible, specially changing modhtmldoc with other more appropriate names. For more information about how *ant* works, we strongly recommend reading the *ant* manual, which is available at http://ant.apache.org**.**

5.3. MODULE EXAMPLES AVAILABLE WITH THE IDK DISTRIBUTION

Our experience in the development of multi-agent systems with the IDK has driven us to implement several modules that are distributed with the IDK for your convenience. They also serve as examples of how to create new modules. Some of these are described in the following sections. For each module, a brief introduction for its purpose is followed by a description of its output (code for a specific platform or verification of properties), its requirements, how to configure it, how to use it, and finally some development issues.

5.3.1 HTML module

This module is used to prepare documentation in HTML format of the specifications.

5.3.1.1 Output of the module

A set of HTML files and image files (for diagrams). One of this file is index.htm from which it is possible to navigate the MAS specification. Diagrams correspond to those defined with the IDK editor.

5.3.1.2 Requirements

This module is integrated with the IDK distribution so there is no requirement for using it.

5.3.1.3 Configuration

No configuration is provided for this module.

5.3.1.4 User manual

The use of this module is fairly easy, just select Modules-> Code Generator-> HTML Document Generator -> generator.

5.3.1.5 Development issues

Changes in the appearance of the generated HTML requires changing the HTML page templates and rebuilding the module as explained in section 5.2.

5.3.2 JADE module

The JADE module supports the translation of INGENIAS interaction diagrams into executable code using JADE Agents communication infrastructures. This code allows to see how those agents would interact and the evolution of their state (a simple application interface is provided for this purpose).

5.3.2.1 Output of the module

In this case, the target platform is the JADE Platform. The code produced is a set of JADE agents with customized behaviours that implement the protocols described with INGENIAS interaction diagrams. A Main class is also generated with the responsibility of launching the generated agents.

5.3.2.2 Requirements

J2SDK 1.4.2. The IDK is distributed with a JADE distribution together with its distribution license (LGPL).

5.3.2.3 Configuration

This platform does not need special configuration. JADE libraries are already included in the distribution. First, you have to compile the JADE module, though it is already compiled in the IDK distribution. To compile the JADE module from the IDK distribution folder, type:

```
ant modJADE
```

Once compiled, start the IDK editor and load an specification like *examples/cinema.xml* (written by Carlos Celorrio and translated to English by Jorge Gómez).

The specification for the JADE platform requires the definition of the following diagrams:

- A protocol, which is defined using an Interaction Diagram and with messages exchange and message ordering.

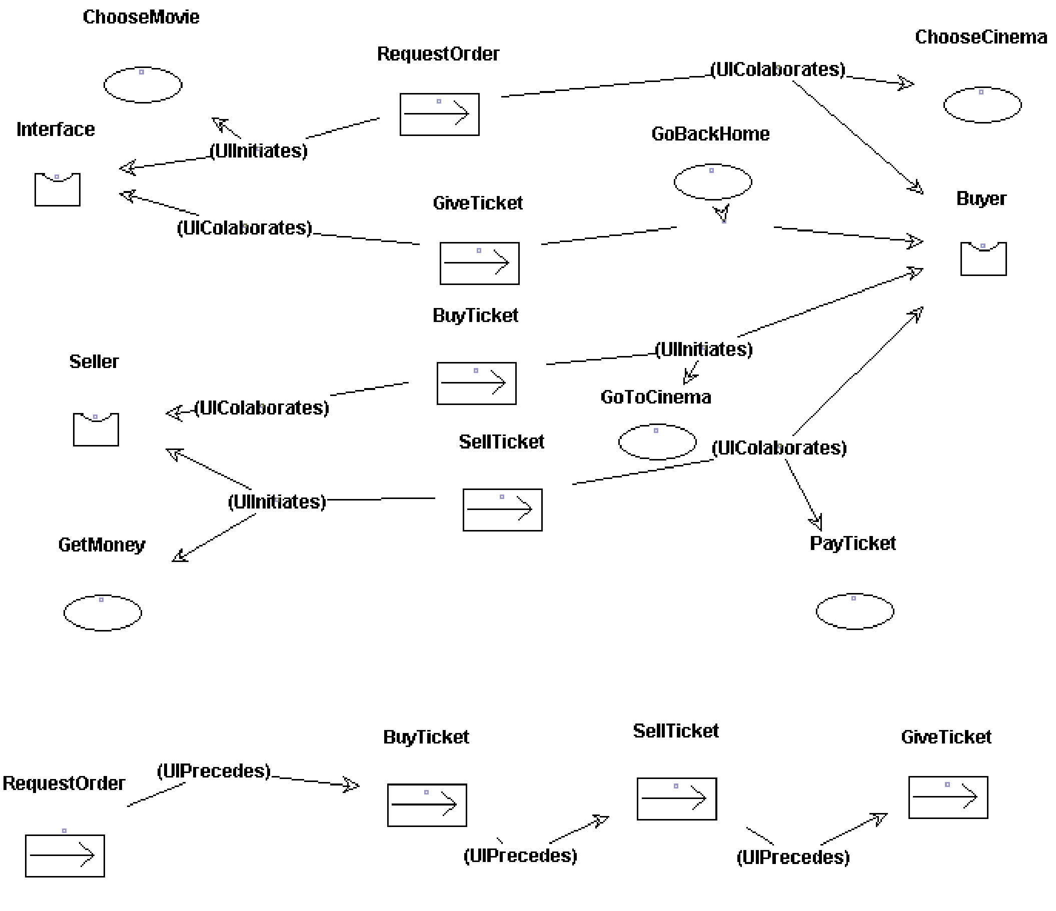
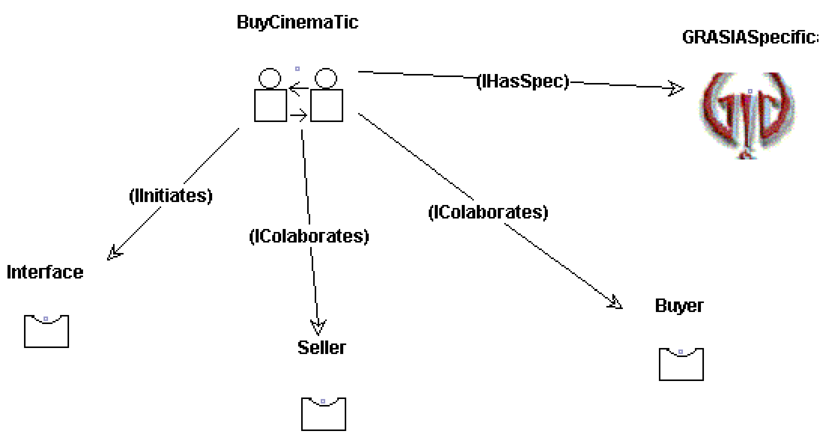
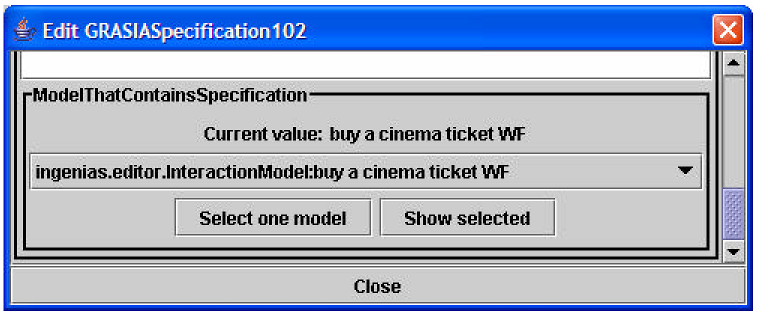


Figure 58. Interaction Protocol definition. Note that there is a separate definition of the messages to be sent (UIInitiates and UIColaborates) and the message order (IUPrecedes)

- Interaction diagrams where Interactions and agents are related (IInitiates and Icolaborates), see Figure 59. It is important also to link a GRASIA Specification entity to the interaction. In this GRASIA Specification, you will have to select the interaction diagram previously defined. Select it in the combo and press *select one* for this.




Edit GRASIASpecification102

ModelThatContainsSpecification

Current value: buy a cinema ticket WF

ingenias.editor.InteractionModel:buy a cinema ticket WF

Figure 59. Interaction diagram defining the Interaction entity, interaction participants, and the specification entity, a GRASIA Specification in this case

- An organization diagram where agents are associated to the roles defined in the interaction

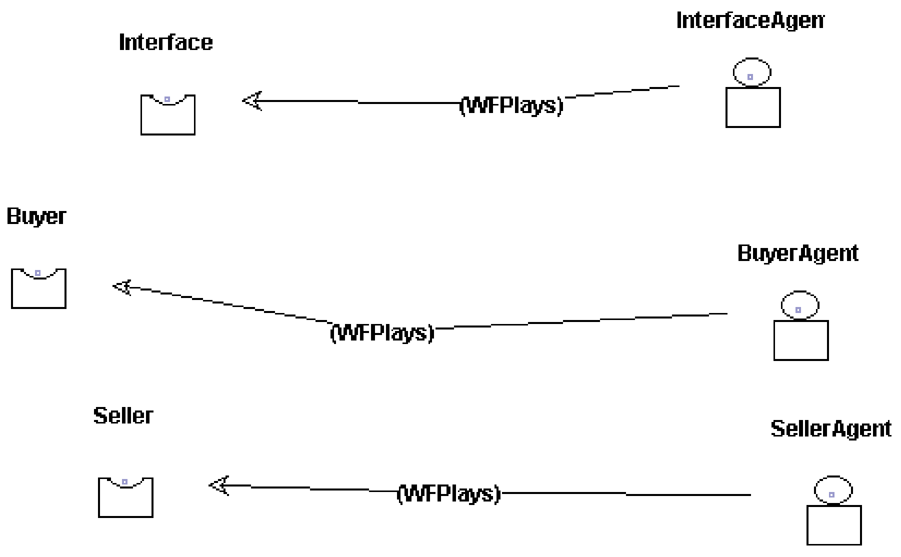


Figure 60. Association of roles participating into the Interaction with their agents

Once defined these diagrams, you can go to the module code generation option and select *generate code* as in Figure 61.

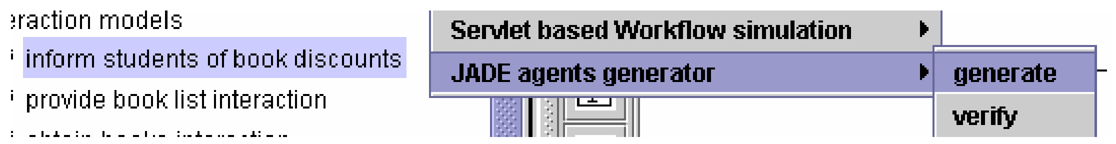

raction models
inform students of book discounts
provide book list interaction
Servlet based Workflow simulation
JADE agents generator
generate
verify

Figure 61. Code generation option for JADE platform

This will produce the source code to run the JADE agents. The next step is to compile the JADE code:

```
ant compJADEmas
```

This will produce the required binaries to run the simulation. Then, to run the simulation, open two consoles in the same IDK distribution folder.

In the first one, type

```
ant runJADE
```

This will start a JADE platform instance. Once launched, you will see the JADE GUI. You have to type in the second console:

```
ant runJADEmas
```

This will open another window with an interface like the one shown in Figure 62.

5.3.2.4 User manual

From the application interface shown in Figure 62 you can start interactions by clicking the buttons on the left part. On the right side you can check state machines allocated in the agents that are participating in the conversations. Each labelled square denote a conversation. Each white square inside the labelled square represents a state machine instance. Above each white square, there is a label indicating which agent owns the state machine. Texts in red represent current states in each state machine.

Figure 62. Snapshot of the main panel that starts individual agent protocols

You can hit any button as many times as you want. The agent will distinguish the different conversation instances of the same or different type.

During the execution, new state machines may appear and make the window size change. Size changes are not handled properly and you may have to resize a little the window to start the layout algorithm and size the frame properly.

Size changes happen because generated agents start protocol state machines under demand. So, you may start with one white panel and see how new panels are created without pressing buttons. The interface window does not close state machine protocol panels when one state machine finishes. To clean a state machine, press *close* button allocated right on the left of the affected state machine panel.

5.3.2.5 Development issues

A current drawback in this implementation is that alternatives in the protocol are not correctly implemented. Right now, if there is a branch in the protocol where the agent has to decide what to do next, by default it is coded that the agent will choose always the first option. However, code implementing other options is generated. Only the decision procedure is missing.

Each agent starts a new conversation only when requested. You can check by running the introspector of JADE and check how many behaviours exist. Also, you can run a sniffer and see what messages are passed.

An agent in this implementation can have many conversations at the same time. The conversation management facilities are built in the *JADEAgent* class and its inheritors.

What to do with this code? The system generates state machines that you can reuse for your system. To allocate the code to execute in each state, look at the if-then sentences inside each state machine specialization.

There are two templates in this module:

- **agent.xml.** This templates defines a *JADEAgent*. Every agent defined in the specification is a JADEAgent and knows as many protocols as the specification says. A JADE Agent contains three main behaviours that are responsible for deciding what protocol to enact according to non-processed messages.
- **ideaprot.xml.** This template defines the state machines that the code generator instantiates according to the role of each agent in an interaction. Each state machine has a controller that determines what is the next state and what message has to be sent.

The most complex part is the traversal of the diagrams to obtain the information required for the specification.

There are two other versions of this module:

- **JADE Leap module.** It is a variation of this module adapted to the JADE Leap platform (see section 5.3.3).
- **JADE Organization.** This one is a more complete version of the module where agent decision procedures are implemented and there are user interfaces to inspect individual agent mental states, task schedulers, and other interesting features (see section 5.3.4).

5.3.3 JADE Leap module

Developed by Carlos Celorrio (ccelorrio@telefonica.net) as a work in the course of software agents of our Ph.D. programme (may 2004). He adapted the JADE module so that it could work with JADE Leap into a PDA (this has been tested on a Sharp Zaurus). Main changes attained the conversion from J2SDK 1.4 to JDK 1.1, modifications of the templates to support task code execution on sending a message, support for agent mobility, and reconfiguration of the JADE module user interfaces.

5.3.3.1 Output of the module

This module generates

1. Agents that can run on a JADE Leap platform.

2. A Main class that launches requested agents.

5.3.3.2 *Requirements*

JDK 1.1.8 (for JADE Leap to run) and J2SDK 1.4.2 (for IDK to run) installed. The JADE Leap platform installed.

5.3.3.3 *Configuration*

In order to run properly the *ant* task that compiles the code generated by the JADE Leap module, you must edit the *leapBuild.xml* file and edit the 'boot-java-1.1-classes' *ant* property so that it points to the *lib/classes.zip* allocated in the distribution folder of JDK1.1.8.

The Demo runs on a PC and a PDA. We assume that the PDA has already a JADE Leap distribution properly installed (please, consult the JADE Leap manual). The PC is a Windows based one, though could be Unix based as well. The PDA in our case is a Sharp Zaurus.

The steps to perform on the PC are:

- Open a command console and go to the IDK distribution folder.
- Deploy the module with:

```
ant modJADEleap
```

- Go to the IDK Editor with:

```
ant runide
```

- Load an specification, for instance the ***cine.xml*** specification allocated in the examples folder of the IDK distribution.
- Go to modules → code generators → JADE LEAP agents generator → generate

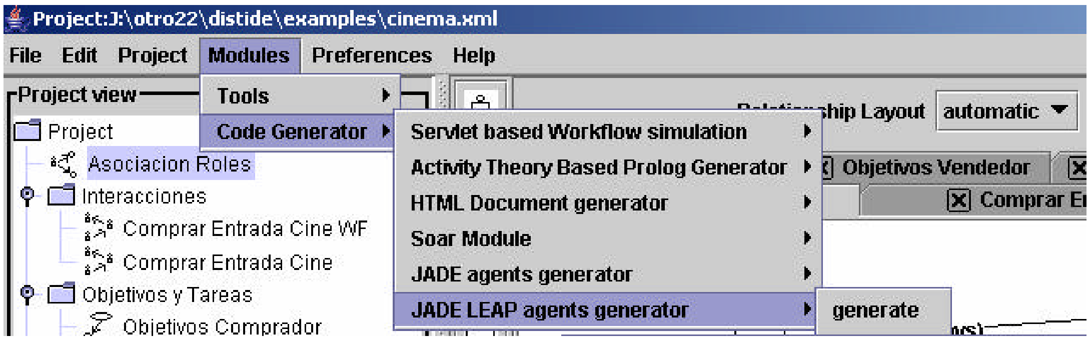
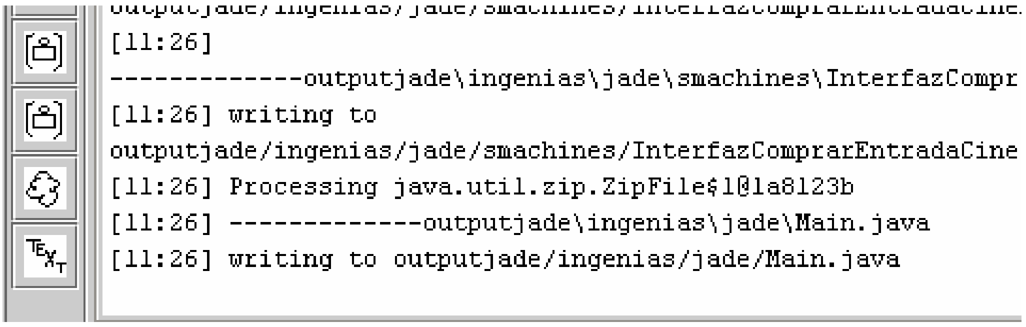


Figure 63. Code generation option for JADE Leap module

- Check that the IDK Editor says that some files were created and written



```
[11:26]
-------------outputjade\ingenias\jade\smachines\InterfazCompr
[11:26] writing to
outputjade/ingenias/jade/smachines/InterfazComprarEntradaCine
[11:26] Processing java.util.zip.ZipFile$1@1a8123b
[11:26] -------------outputjade\ingenias\jade\Main.java
[11:26] writing to outputjade/ingenias/jade/Main.java
```

Figure 64. Produced messages on generating code for JADE Leap

- Close the IDK Editor
- Edit the *build.xml* file and set the property *boot-java-1.1-classes* to the *classes.zip* file allocated in the JDK1.1.8 distribution.

- Compile the generated code with:

```
ant compJADEleappjava
```

- Upload the generated jar ***genlib/demoPjava.jar*** into the PDA. This step depends on the software that allows to transfer files from the PC and the PDA. Sometimes, it is a desktop application that connects directly with the PDA; others, it is a matter of transferring with ftp the file to the PDA or dragging and dropping it into a special *My PC* drive.

- Start Jade Leap Main Container with:

```
ant runJADEleap
```

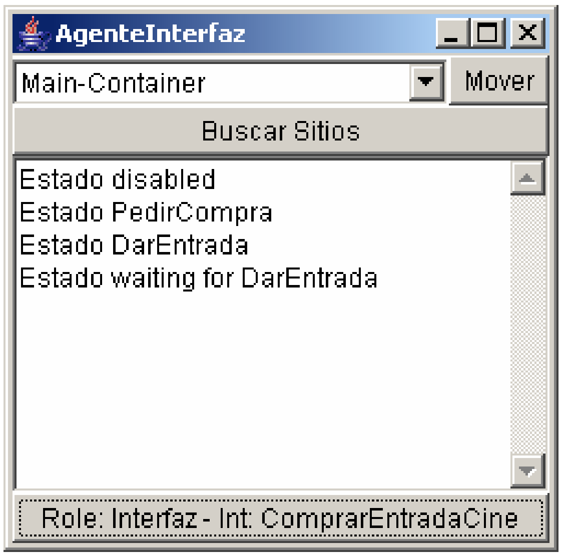
In the PDA (once the Main Jade Leap is running and the code of the agents code (the previously mentioned jar file) is uploaded into the PDA:

- Start Jade Leap Agents within the PDA. In the ZAURUS, this is done with the JVM implementation named *evm*

```
evm –cp demoPjava.jar:JadeLeap.jar ingenias.JADE.Main –container –host 192.168.129.1
```

The "-host" parameter may be changed because it is the IP location of the Jade Leap Main Container (that must be running in the PC). Zaurus USB Lan uses "192.168.129.1" to identify the PC.

- Agent's GUIs should appear in the PDA, like the one from Figure 65



```
Estado disabled
Estado PedirCompra
Estado DarEntrada
Estado waiting for DarEntrada
```

Figure 65. GUI representing an agent

- Move Vendor Agent to the PC by selecting 'Main-Container' in its GUI location selector and clicking 'Move'.

- To start the interaction click on the Interaction Button placed in the bottom of the Interface Agent's GUI.

5.3.3.4 User manual

When the agents are started as described above, there will be a window for each agent that is running. The agent GUI has three different zones (Figure 65):

- In the top of the window there is a selector to choose among the locations the mobile agent can go. Also, there is a button named 'Move' to go to the selected location.

- In the middle of the window there is a big text panel where it will be displayed the states that each agent transits.

- Every agent who does start an interaction (the interaction initiator) will have a button in the bottom of the window. When that button is clicked, the interaction will initiate. There will be a button for each interaction and it is possible for an agent to have more than one button.

The state machine that governs the behaviour of the agents is the same as in the JADE Module.

5.3.3.5 Development issues

This module is based on the INGENIAS JADE Module. It has been adapted to compile with a JDK1.1 distribution, so that the code is suitable to be executed on every PC as well as on any PDA that complies with the Personal Java Profile.

The Swing GUI has been replaced with an AWT GUI, for the same reasons. The state of the agents is no more presented graphically in a single window, but in a textual way on the agent's GUI window.

On enabling mobility so that the agents could jump from one container to another, there were some problem serializing the agent's GUI. Because of this, it was decided to make it transient (not serializable) and it is disposed and reconstructed every time the agent moves to a different location, so it is important that the JADEAgentGUI classes are in the classpath in the target location where the agent moves. Nevertheless, the rest of the agent classes, including its JADE behaviours and state machines, are correctly serialized and sent to the target machine.

5.3.4 JADE Organization module

Developed by Jorge J. Gómez Sanz. The JADE organization module is a continuation of the effort of the JADE module. This new module provides computational representations for tasks as well as improvements for the protocol implementation.

- Protocol improvements
  - Multiple states in the state machines that codify the behaviour of each interaction part.
  - Transitions in the state machine are guided by conditions defined in the specification. Conditions refer to the presence of specific facts in the mental state of the agent and the satisfaction of conditions defined by the developer.
  - Automatic evolution of the protocol. The information interchange across agents stops at certain points until a decision is made about what to do next. The execution continues when the information required by the protocol to continue appears in the mental state of the agent. This way, the protocol and the agent control are somehow decoupled.
- Task computational representation
  - A task will be eligible for execution if and only if it can satisfy any of the current existing goals
  - Every time a task is executed, it produces facts that are asserted in the mental state of the executor
  - If according to a workflow definition, a task executed in one agent generates information required by a task into another agent, this information is forwarded automatically from agent to agent if and only if both tasks appear in a interaction diagram
  - A task is initiated if and only if there exist the inputs it requires
  - Internally, each tasks has access only to the resources assigned and the mental entities associated to it in the specification

Resulting MASs are almost functional. From the abstract architecture that defines the INGENIAS metamodel, it remains to be implemented the Mental State Processor and the Mental State Manager. In every MAS generated with this module, the functionality associated with these elements is assigned to the user. The user can decide which task to execute and what to do with each mental state entity by means of a graphic user interface.

This module has been developed using the Juul Bokhandle case study as base. Readers can review this case study in section 6.2.

5.3.4.1 Output of the module

Agents that can run on a JADE Leap platform. A Main class that launches requested agents.

5.3.4.2 Requirements

JDK 1.5.0 Update 4 installed. JADE is required as well, but it is included within the IDK

5.3.4.3 Configuration

All libraries needed to compile the module are already included in the distribution

```
ant modjadeorg
```

Once compiled, it will be uploaded automatically by the IDK. Remember that the IDK supports hot deployments, i.e. you can compile modules while the IDK is running.

To define a MAS system to be processed by this module the following diagrams have to be defined:

- **An agent diagram per agent.** This diagram will show each agent's initial mental state and pursued goals.

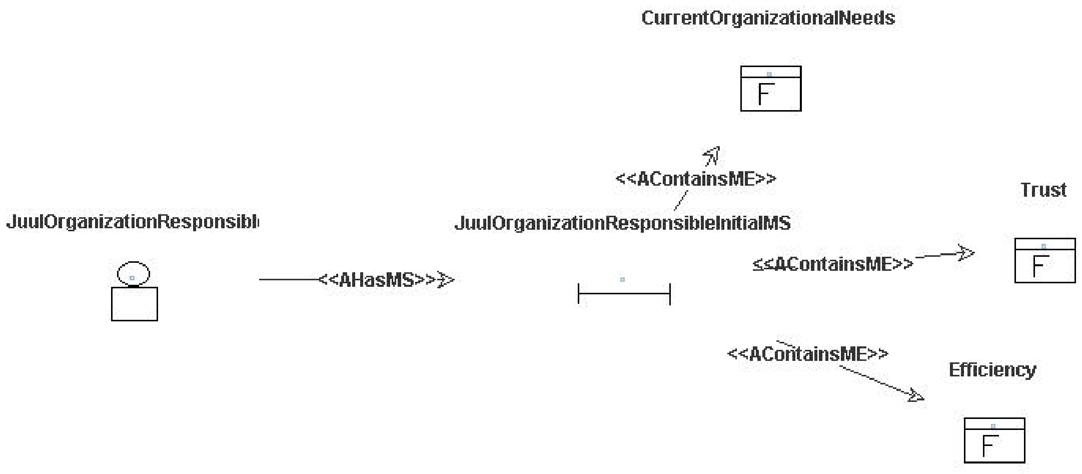

CurrentOrganizationalNeeds
<<AContainsME>>
Trust
JuulOrganizationResponsibl
JuulOrganizationResponsibleInitialMS
<<AHasMS>>
<<AContainsME>>
<<AContainsME>>
Efficiency

Figure 66. Initial mental state for the responsible of the organization. The diagram states that the responsible of the organization Juul starts with three facts in its mental state.

- **An organization diagram showing the workflow of tasks.** The workflow is represented by means of a set of tasks connected with WFConnects relationships. These relationships means that the output of a task is used as input of another. In these diagrams, responsibility relationships (WFResponsible) from roles to tasks should be defined as well.

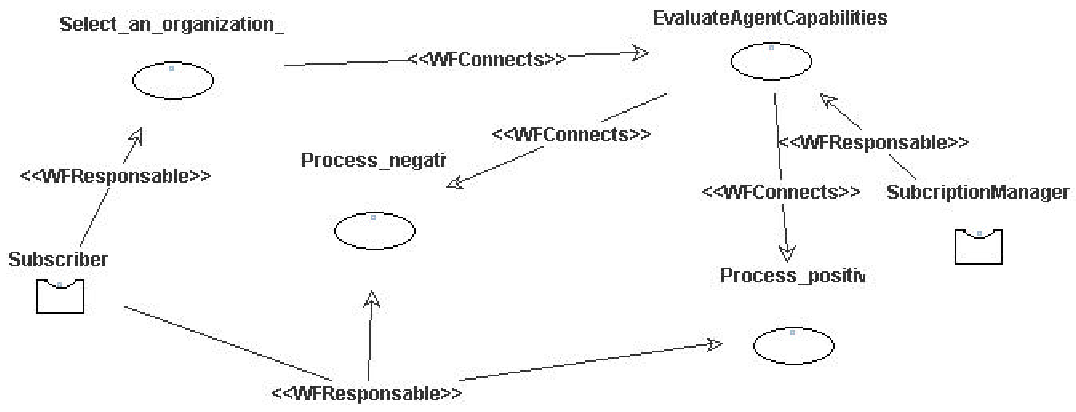


Figure 67. Representation of the workflow of the tasks in the system. *WFConnects* relationships represent input-output dependencies among tasks. *WFResponsable* denote responsibility relationships. Note that we use only roles and not agents. This is a requirement of the module.

- **Goal tasks diagrams for each task and goal.** Each goal defined in the system has to be associated with

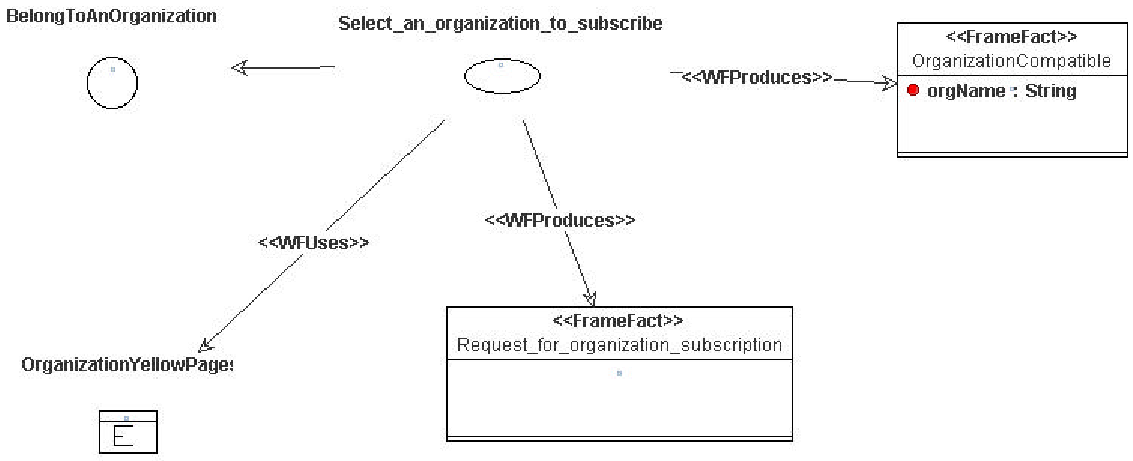


Figure 68. Description of the *task select an organization to subscribe to. WFConsumes* relationship indicates which elements has to present in order to initiate the task. Also, the task is linked with a goal to indicate that it can produce the elements required to satisfy the goal.

- **An interaction diagram per interaction.** The interaction is represented by an interaction entity in which several roles participate. Direct association of agents to interactions is not supported. Instead, use a role and then associate the role with each agent in an organization diagram

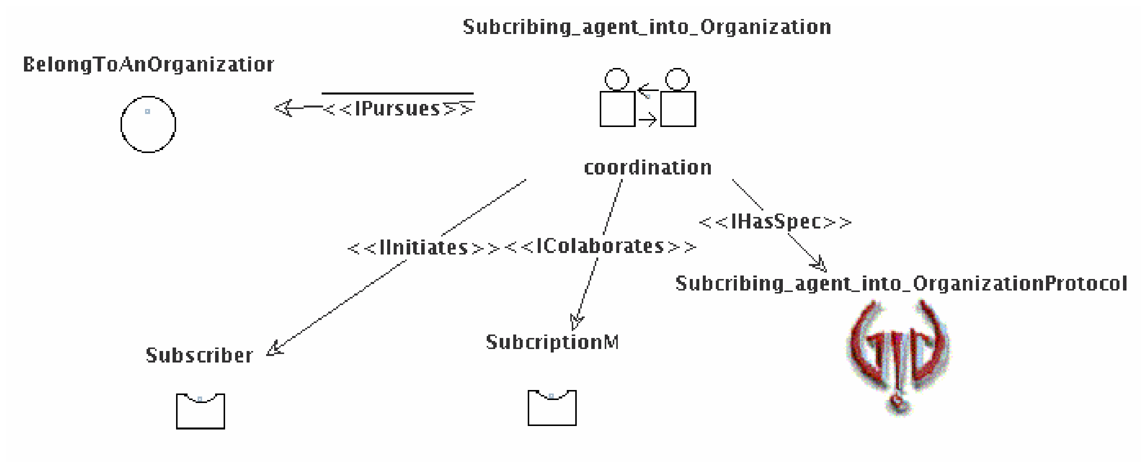


Figure 69. High level representation of the interaction. The interaction pursues the goal *belong to an organization* and is specified into a separated protocol with the GRASIA notation. The protocol is detailed in Figure 70. Participants of the interaction can be only roles. There have to be exactly one initiator and at least one colaborator.

- **An interaction diagram showing the protocol.** By now, we still use grasia protocol representations. AUML diagrams are not supported yet. During the representation of the protocol, it will be required that you

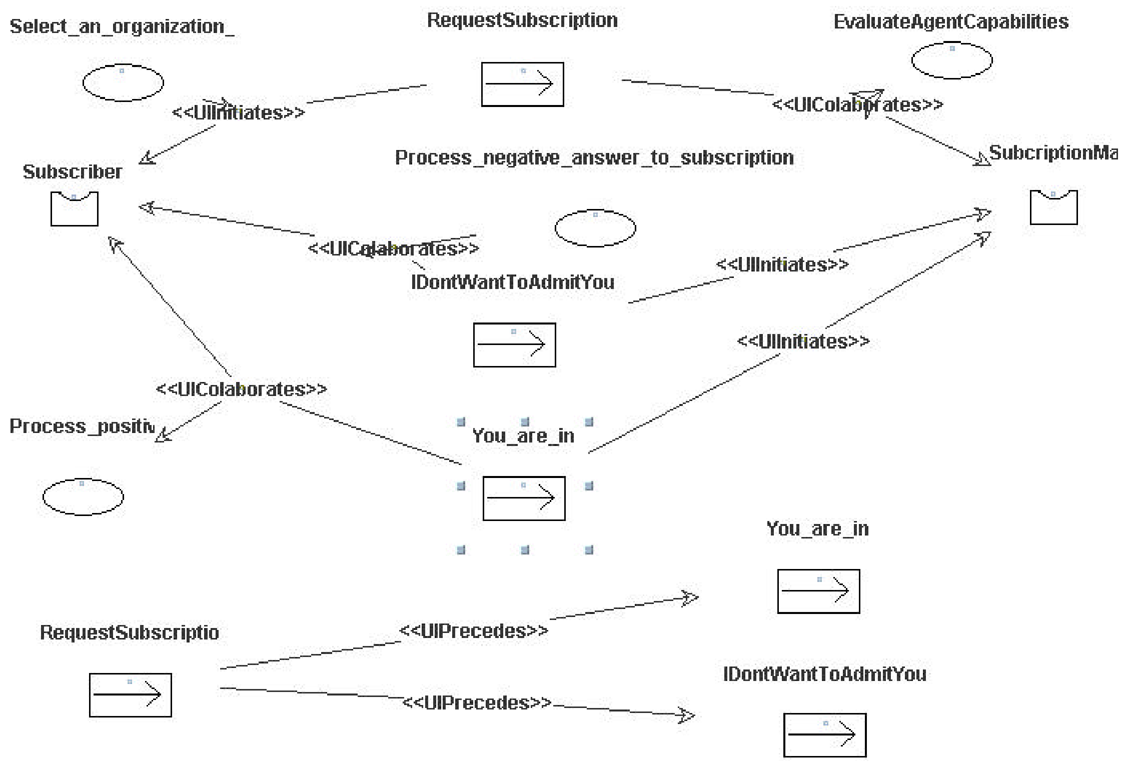


Figure 70. Protocol representation for the interaction *subscribing agent into organization*, which is represented in Figure 69

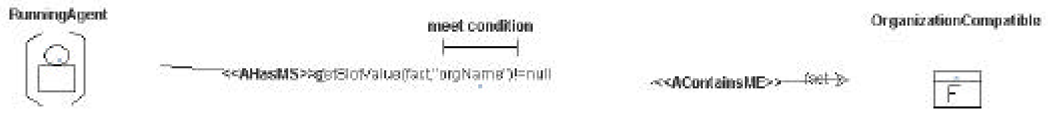


Figure 71. Mental state required to initiate a request subscription interaction unit

Once defined these diagrams, it is required to proceed as in the case of the JADE module: generating the code as Figure 72 remarks.

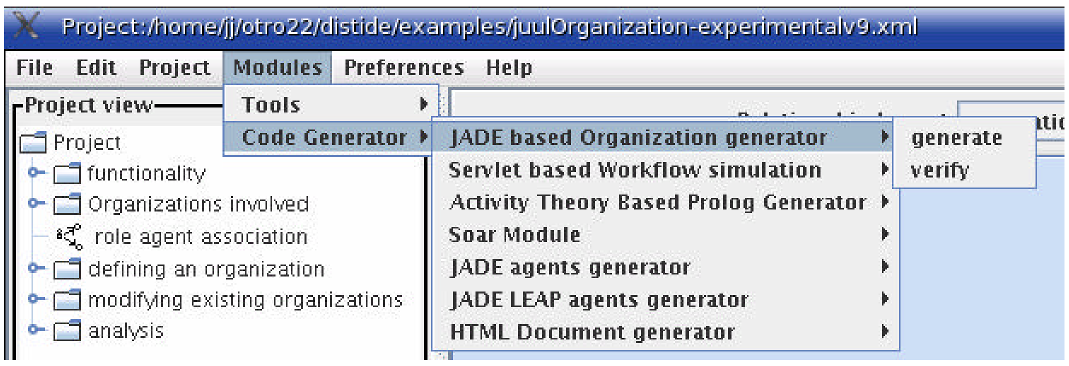

Project:/home/jj/otro22/distide/examples/juulOrganization-experimentalv9.xml
File
Edit
Project
Modules
Preferences
Help
Project view
Project
functionality
Organizations involved
role agent association
defining an organization
modifying existing organizations
analysis
Tools
Code Generator
JADE based Organization generator
Servlet based Workflow simulation
Activity Theory Based Prolog Generator
Soar Module
JADE agents generator
JADE LEAP agents generator
HTML Document generator
generate
verify

Figure 72. Selecting code generation for JADE Organization module

During code generation several warnings can be raised in the messages panel:

```
[02:39] WARNING: In Interaction unit notify_students_of_book_discounts the collaboration relationship does not remark a mental condition. If you feel that this collaboration requires a mental condition, just double click on the relationship
```

This warning states that in an interaction unit, the collaborator is not describing any guards for defining conditions that enable participation. The effect is that no matter what is happening, the collaborator will process the message and the information it contains will be asserted into the agent mental state.

```
[02:39] WARNING:There are no FRAME FACTS defined in diagram diagram mental condition to collaborate with request subscription. Interaction Subcribing_agent_into_Organization in interaction unit RequestSubscriptionsays that this diagram defines a mental state condition and facts may be required. Please, check that you are using FrameFacts. To do so, select a fact that you are using and right click on it, and check that there is no any refine option. If there is, select in the refine submenu the FrameFact entry
```

This time, we have defined an empty mental state condition in a interaction diagram. The message explains how to solve it.

If the developer considers convenient ignoring these warnings, the next step is compiling the output. This is executed with an ant command

```
ant compjademasorg
```

Now it is time to open another console within the same folder. In one, you have the JADE platform

```
ant runjade
```

In the other, you will start the main GUI to handle the resulting MAS

```
ant runjademasorg
```

5.3.4.4 User manual

For running a demo, we recommend using the file ***juulOrganization-experimentalv9.xml*** file, which is included in the examples folder of the IDK distribution. To run the complete demo, the user should have:

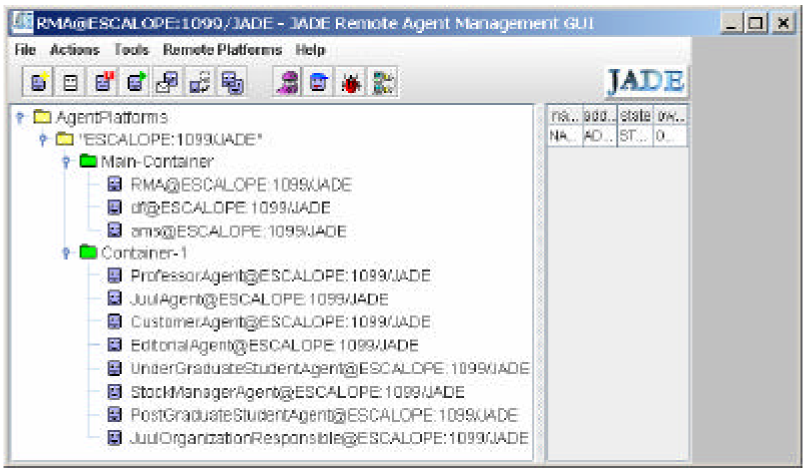
1. Compiled the jade organization module, if not compiled yet. By default, the IDK includes already compiled modules.
2. Loaded the specification with the IDK
3. Generated the code with the module jade organization, see Figure 72.
4. Compiled the generated code
5. Run the jade platform in one console
6. Run the MAS in another console

We assume that all these steps, described in section 5.3.4.3, have been completed successfully. If so, the user should have two screens presented here in Figure 73 and Figure 74. Figure 73 represents the management GUI of the JADE platform once all the MAS agents have been started. They appear under the *container-1* folder. By now, we will center the discussion into two of them: the JuulAgent and JuulOrganizationResponsible.

Figure 74 represents the GUI where end-users interact with their agents. It is composed of three sections:

- **Interaction Management.** Not all agents can start an interaction, and not all interactions can be started inmediately. Figure 74 shows at the left names of agents with buttons representing interactions they can initiate. By clicking on one button, the agent will locate proper participants (those who play the roles described in the specification), and will start the conversation, provided that mental conditions hold. These mental conditions would be those associated to *UIInitiates* relationship instances, such as the one presented in Figure 71. Once started the interaction continues until reaching a point where certain mental state conditions do not hold. At that moment, the interaction will stop, but will be resumed when the mental state changes. During the execution of the interaction, the user will observe state machines evolve, as in XXXX. The red nodes represent current activated interaction states. If the interaction does not progresses, the reason may be that it is waiting a task to produce certain mental entities.

- **Task Management.** Figure 76 presents the GUI used to interact with tasks. At any moment, only those tasks whose execution satisfies a goal are shown. This is related with the Rationality Principle and BDI concerns. By clicking on the radion button situated at the right side of each tasks, we can inspect what kind of inputs it is expecting and what outputs. The representation of inputs and outputs is achieved by means of agent diagrams, since these are the most adequate to represent mental entities. Since we are handling agent diagrams, exactly the same used in the IDK, we are allowed to edit entities, move them, or resize. Deletion or aggregation of new ones will not have any effect in this version, though the interface allows to execute these operations. This way, a user can manually modify the behavior of a task without altering its code. When the user decides to execute a task, all expected inputs have to be present in the mental state of the agent, otherwise an error message will be triggered. If executed, the user can check the agent mental state and verify that new entities have been added.
- **Mental State Management.** Figure 77 shows the GUI for monitoring the agent mental state. As in the task management GUI, the user can resize, move, or edit existing entities. Also, deleting or adding new entities will not have any effect, since the diagram reads the agent mental state but is not able to write on it, though it provides modification operations of already defined entities.



```
AgentPlatforms
  "ESCALOPE:1099/JADE"
    Main-Container
      RMA@ESCALOPE:1099/JADE
      df@ESCALOPE:1099/JADE
      ams@ESCALOPE:1099/JADE
    Container-1
      ProfessorAgent@ESCALOPE:1099/JADE
      JuulAgent@ESCALOPE:1099/JADE
      CustomerAgent@ESCALOPE:1099/JADE
      EditorialAgent@ESCALOPE:1099/JADE
      UnderGraduateStudentAgent@ESCALOPE:1099/JADE
      StockManagerAgent@ESCALOPE:1099/JADE
      PostGraduateStudentAgent@ESCALOPE:1099/JADE
      JuulOrganizationResponsible@ESCALOPE:1099/JADE
```

Figure 73. JADE platform started with the demo

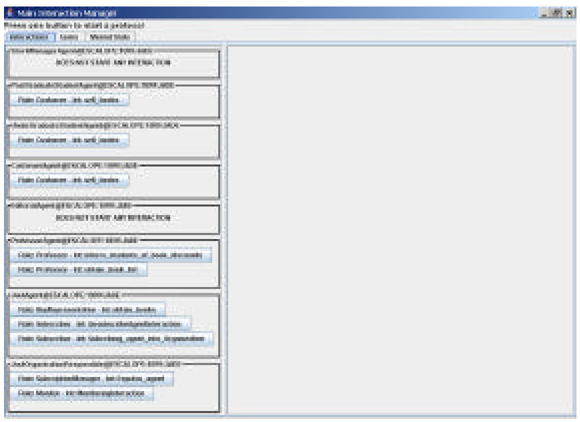
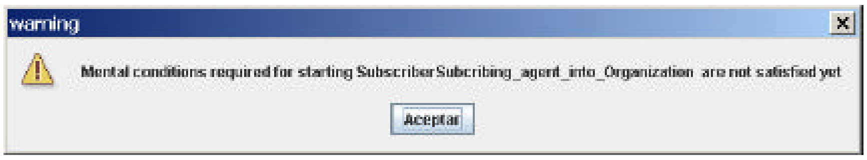


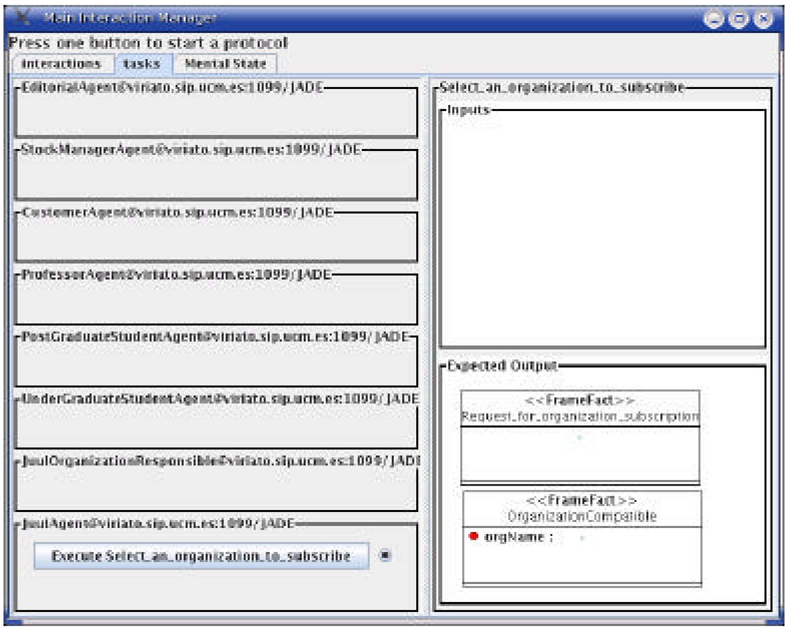
Figure 74. Main GUI for agent-user interaction


warning

Mental conditions required for starting SubscriberSubcribing_agent_into_Organization are not satisfied yet

Aceptar

Figure 75. When the mental conditions defined in the interaction do not hold, an alert message is shown


Main Interaction Manager

Press one button to start a protocol

interactions | tasks | Mental State

EditorialAgent@viriato.sip.ucm.es:1099/JADE

StockManagerAgent@viriato.sip.ucm.es:1099/JADE

CustomerAgent@viriato.sip.ucm.es:1099/JADE

ProfessorAgent@viriato.sip.ucm.es:1099/JADE

PostGraduateStudentAgent@viriato.sip.ucm.es:1099/JADE

UnderGraduateStudentAgent@viriato.sip.ucm.es:1099/JADE

JuulOrganizationResponsible@viriato.sip.ucm.es:1099/JADE

JuulAgent@viriato.sip.ucm.es:1099/JADE

Execute Select_an_organization_to_subscribe

Select_an_organization_to_subscribe

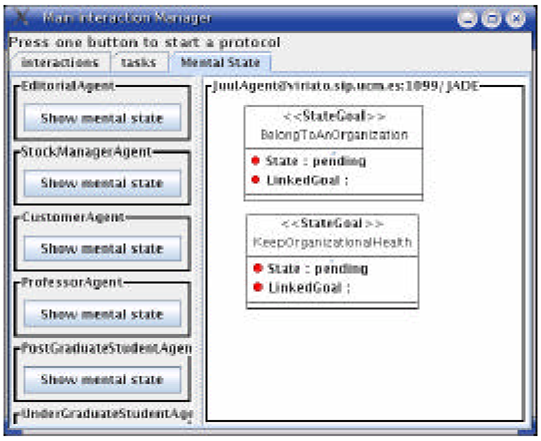
Inputs

Expected Output

<<FrameFact>>
Request_for_organization_subscription

<<FrameFact>>
OrganizationCompatible
- orgName :

Figure 76. Task management GUI. It shows for the current selected task, the one with his radio button activated, the expected inputs there should exist and the outputs that should be produced. The name of the task appears in the external border of the right panel. The upper part of the right panel contains the inputs and the bottom one, the outputs.


Main Interaction Manager

Press one button to start a protocol

interactions | tasks | Mental State

EditorialAgent
Show mental state

StockManagerAgent
Show mental state

CustomerAgent
Show mental state

ProfessorAgent
Show mental state

PostGraduateStudentAgen
Show mental state

UnderGraduateStudentAge

JuulAgent@viriato.sip.ucm.es:1099/JADE

<<StateGoal>>
BelongToAnOrganization
- State : pending
- LinkedGoal :

<<StateGoal>>
KeepOrganizationalHealth
- State : pending
- LinkedGoal :

Figure 77. Initial mental state associated to Juul Agent. It shows two goals and the name of the agent in the border of the right panel.

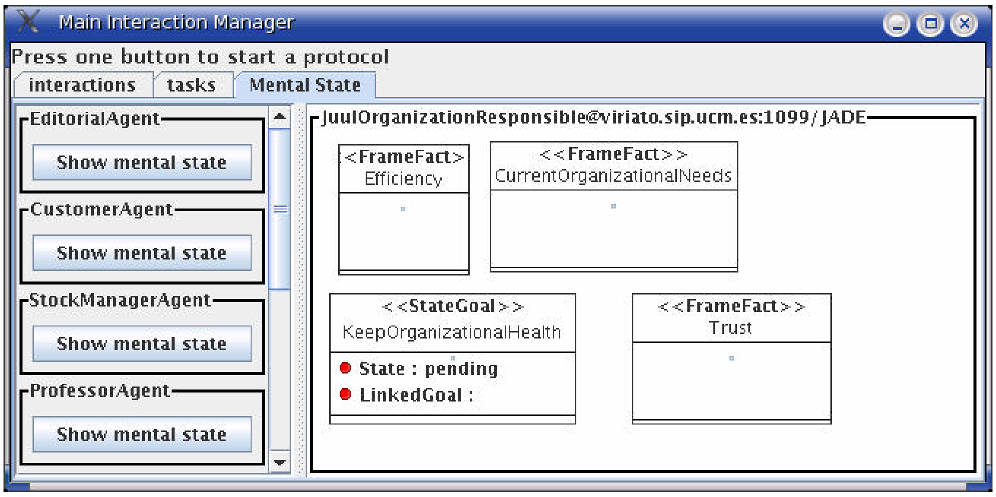


Figure 78. Initial mental state of agent Juul Organization Responsible

To demonstrate the capabilities of the module, we will show the different steps in executing the *subscribing agent into organization*, presented in Figure 69 and Figure 70.

Step 1. We go to the task manager and press on **execute select an organization to subscribe**. This button is shown in Figure 76. To view the result, switch to the mental state manager and watch the mental state of agent Juul by pressing its **show mental state** button. You should see what Figure 79 shows.

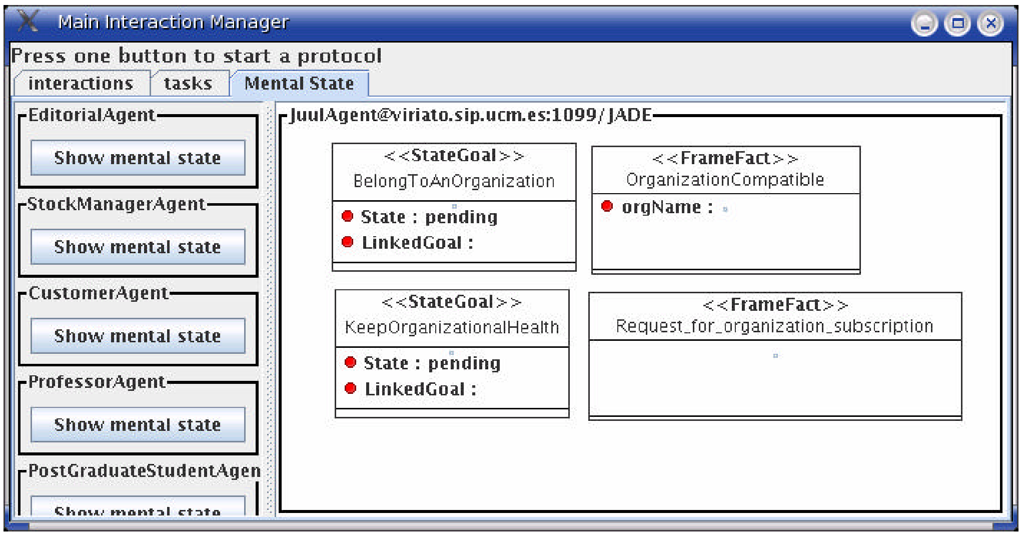


Figure 79. Mental state of the Juul Agent after executing task *select an organization to subscribed,* presented in Figure 76.

Step 2. Go to the interactions manager and press the button with label **Role: Subscriber – Int: subscribing agent into organization**. After pressing it, new panels will appear in the right side. The first describe the states recognised by the subscriber, and the second the states recognised by the subscription manager, as shows Figure 80. The protocol will evolve until there is some step that requires certain information to be present. This information is defined with diagrams such as Figure 71 and it is associated to the *UIInitiates* or *UIColaborates* relationship by double clicking on them in the IDK. In this case, the communication progresses until reaching a division of the communication flow: whether to accept the subscription request or not. This cannot be decided yet, since the agent needs to evaluate the proposal. Hence, that both protocol state machines, the one from the sender and the receiver, are currently into two states: it waits for both confirmation or the rejection. A secondary effect of the execution of the interaction is the aggregation of new mental entities to the mental state of the receiver, the Juul Organization Responsable. Figure 81 presents the mental state of this agent. If you compare it with the previous one Figure 78. The information sent is the one produced by task *select an organization* *to subscribe to*, presented in Figure 68. It is sent because this task appears linked to the UIInitiates relationship, shown in Figure 70.

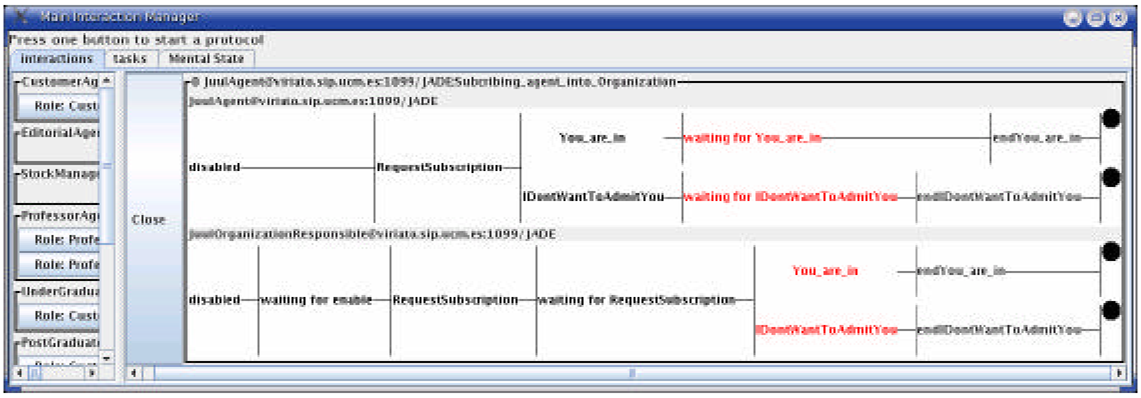


Figure 80. Interaction *subcribing agent into organization* in the middle of its execution. The different words represent states in the interaction. Nodes in red represent current states. The notation is petri-net like. Black circles represent end states. Initial states are those located at the left hand side.

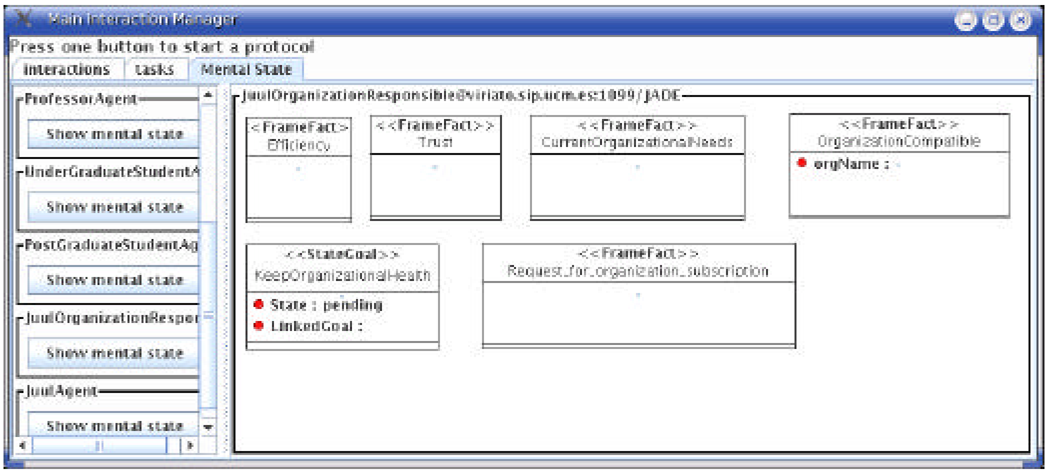


Figure 81. As a result of the interaction, some information has been sent to the receiver, the Juul Organization Responsible Agent. You can review the new information by pressing on its button.

Step 3. Now it is time to decide whether if Juul Organization Reponsible agent accepts Juul proposal or not. According to Figure 84 and Figure 85, we need to set a field of a the *Evaluation Result* fact to either ***accepted*** or ***rejected***. To do so, we select the task *evaluate agent capabilities* in the task manager, as shows Figure 82. Now it is required to edit the *evaluation result* as Figure 83 illustrates. Once modified, we press the *execute evaluate agent capabilities* button

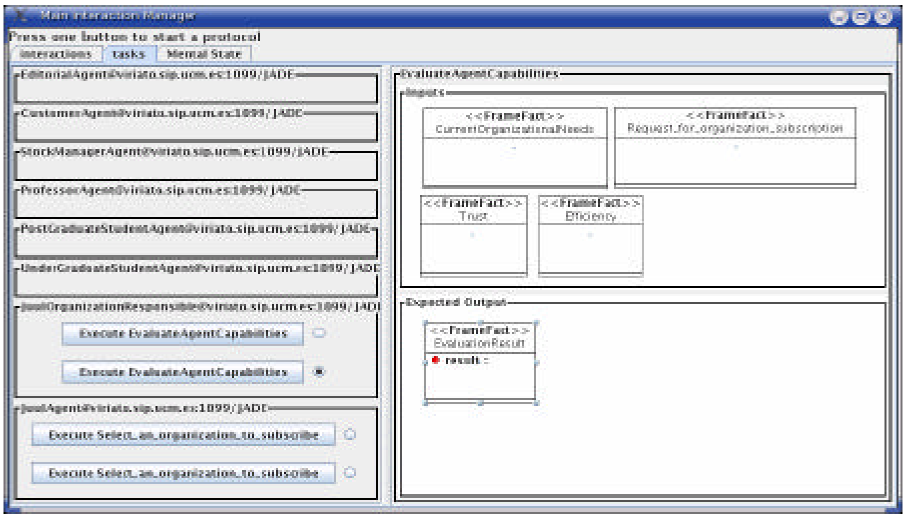

Main Interaction Manager
Press one button to start a protocol
interactions
tasks
Mental State
EditorialAgent@viriato.sip.ucm.es:1099/JADE
CustomerAgent@viriato.sip.ucm.es:1099/JADE
StockManagerAgent@viriato.sip.ucm.es:1099/JADE
ProfessorAgent@viriato.sip.ucm.es:1099/JADE
PostGraduateStudentAgent@viriato.sip.ucm.es:1099/JADE
UnderGraduateStudentAgent@viriato.sip.ucm.es:1099/JADE
JoulOrganizationResponsible@viriato.sip.ucm.es:1099/JADE
Execute EvaluateAgentCapabilities
Execute EvaluateAgentCapabilities
JoulAgent@viriato.sip.ucm.es:1099/JADE
Execute Select_an_organization_to_subscribe
Execute Select_an_organization_to_subscribe
EvaluateAgentCapabilities
Inputs
<<FrameFact>> CurrentOrganizationalNeeds
<<FrameFact>> Request_for_organization_subscription
<<FrameFact>> Trust
<<FrameFact>> Efficiency
Expected Output
<<FrameFact>> EvaluationResult
result :

Figure 82. Activation of a new task, the *evaluate agent capabilities* task. We need to edit the *Evaluation Result* fact in order to modify its default value

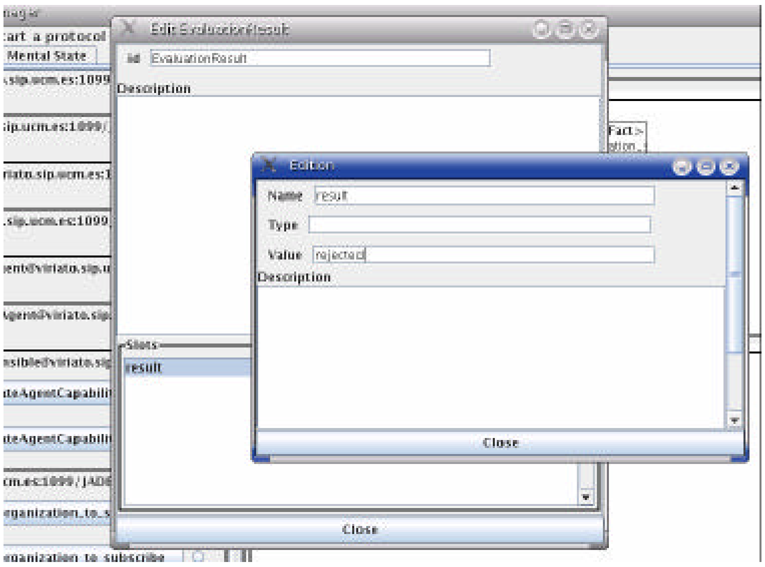


Figure 83. Editing the *evaluation result* fact. Edition is applied as in is done in the IDK. First you double click the fact. In the resulting dialog window, you scroll down, select the entry *result* in the list, right click on it and select open. Once opened, write down in the *value* field either ***rejected*** or ***accepted***.

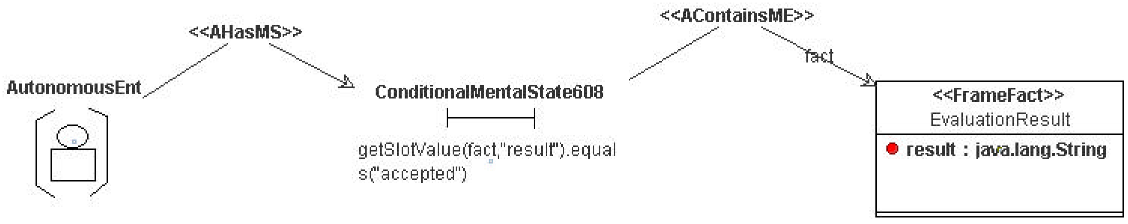

<<AHasMS>>
AutonomousEnt
ConditionalMentalState608
getSlotValue(fact,"result").equals("accepted")
<<AContainsME>>
fact
<<FrameFact>>
EvaluationResult
result : java.lang.String

Figure 84. Mental state condition required to initiate the *You are in* interaction unit in Figure 70.

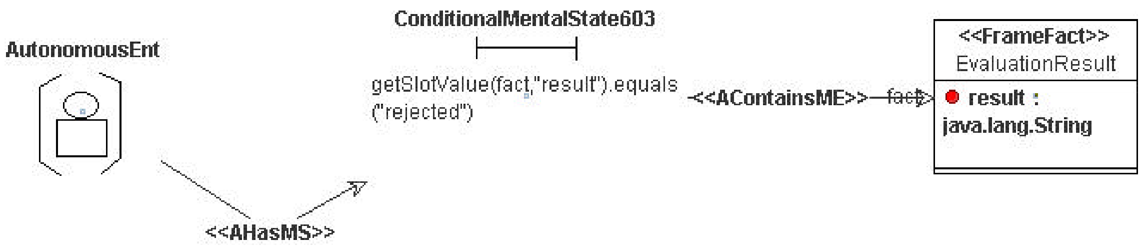

AutonomousEnt
ConditionalMentalState603
getSlotValue(fact,"result").equals ("rejected")
<<AContainsME>>
fact
<<FrameFact>>
EvaluationResult
result : java.lang.String
<<AHasMS>>

Figure 85. Mental state condition required to initiate the *I don't want to admit you* interaction unit in Figure 70

Step 4. There is little more to do. Since the state machines representing the procotols have automatically self activated, the interaction manager will show you some changes in the original state machines presented in Figure 80. Figure 86 contains the final state reached. The state machine for agent Juul Agent has reached two final states. This happen due to the protocol definition. According to how it was defined, there is no way to distinguish between the executing of a *I don't want you* and a *you are in* interaction units without inspecting the *evaluation result* fact sent by Juul Organization Responsible Agent. Unfortunately, *evaluation result* cannot be known by Juul Agent until it is asserted into its mental state. Nevertheless, if it is asserted, then it can be said that the interaction unit has been executed successfully. The solution would be to define conditions to be applied not only to the agent mental state, but to the interaction unit iself.

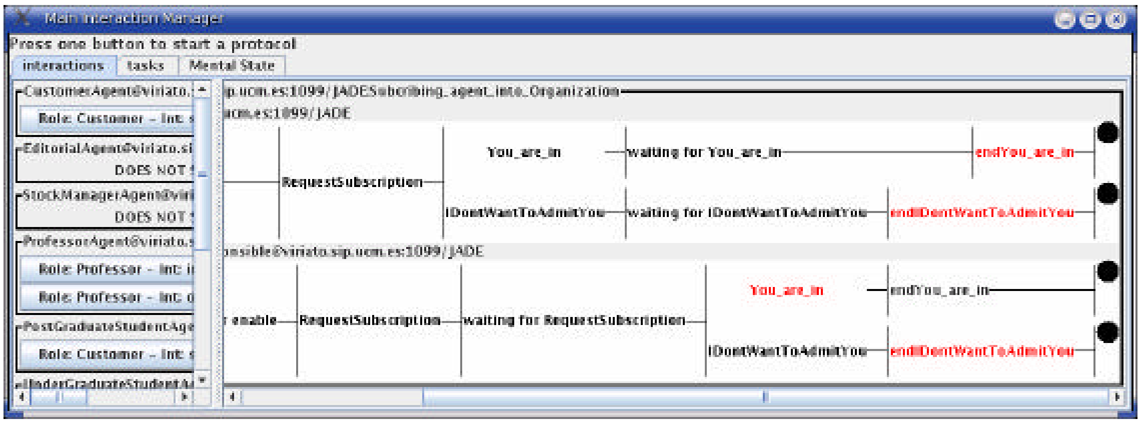


Figure 86. Interaction final state after introducing the *evaluation result* fact in the Juul Organization responsible agent.

The effect of the interaction can be observed in the mental state of Juul Agent, see Figure 88. Also, by finalizing the interaction, new tasks are enabled, as shows Figure 87. These new tasks are activated due to the goal *keep the organizational health*, and the presence of the fact *evaluation result.*

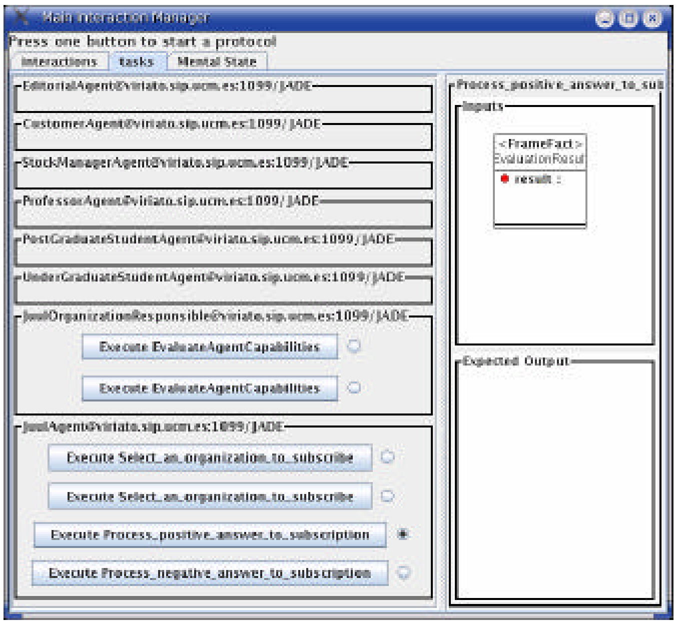

Main Interaction Manager
Press one button to start a protocol
interactions
tasks
Mental State
EditorialAgent@viriato.sip.ucm.es:1099/JADE
CustomerAgent@viriato.sip.ucm.es:1099/JADE
StockManagerAgent@viriato.sip.ucm.es:1099/JADE
ProfessorAgent@viriato.sip.ucm.es:1099/JADE
PostGraduateStudentAgent@viriato.sip.ucm.es:1099/JADE
UnderGraduateStudentAgent@viriato.sip.ucm.es:1099/JADE
JuulOrganizationResponsible@viriato.sip.ucm.es:1099/JADE
Execute EvaluateAgentCapabilities
Execute EvaluateAgentCapabilities
JuulAgent@viriato.sip.ucm.es:1099/JADE
Execute Select_an_organization_to_subscribe
Execute Select_an_organization_to_subscribe
Execute Process_positive_answer_to_subscription
Execute Process_negative_answer_to_subscription
Process_positive_answer_to_sub
Inputs
<<FrameFact>>
EvaluationResul
result :
Expected Output

Figure 87. New activated tasks in agent Juul Agent after receiving the *Evaluation Result* fact.

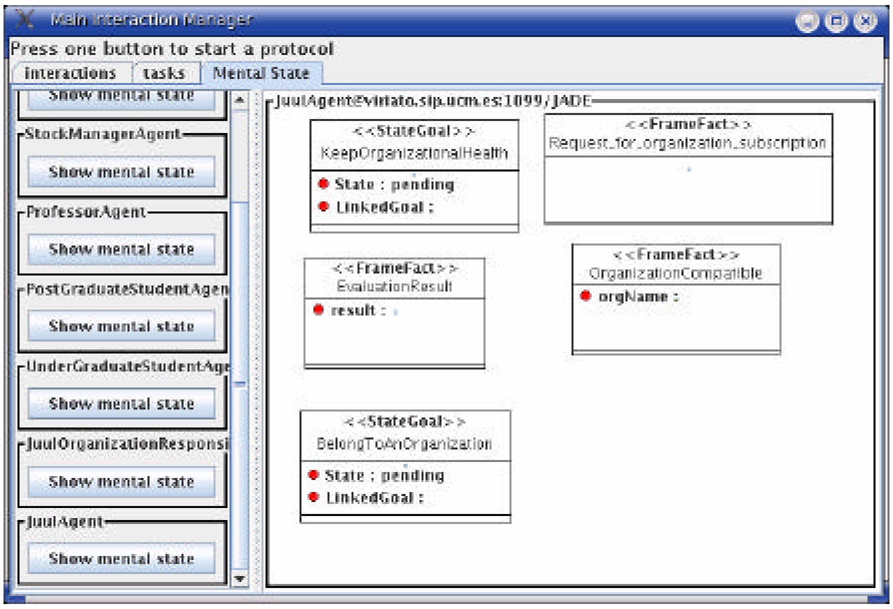


Figure 88. Mental state of Juul Agent after finalizing the interaction. Note that there is a new fact, the *evaluation result* fact.

5.3.4.5 Development issues

There are some open issues in this module:

[23/08/2005]

- Sometimes the GUI hides the fields of entities shown. So instead of showing elements as in Figure 77, the name of the entity and its fields are hidden.
- The state machines show a quite slow transitions. This is programmed on purpose in order to facilitate visual inspection of the protocols
- By now, it is not possible to define mental state conditions that depends on the information sent by another agent.
- Dynamics of goals and other mental entities is not complete. Goals exist forever and they serve mainly as task activators. No mental entity is removed from the mental state,

what causes problems when trying to repeat an interaction. When repeating an interaction, all facts required to make it progress are already present, so it would continue without control.

- Facts in the mental state manager and in the task manager do not show their default values

5.3.5 SOAR module

Developed by Juan Antonio Recio (jareciog@ucmail.ucm.es) as a work in the course of software agents of our Ph.D. programme (may 2004). It provides a mapping from INGENIAS to SOAR code. An example is provided of agents playing the role of tanks.

5.3.5.1 Output of the module

SOAR rule specification files detailing the behaviour of a tank.

5.3.5.2 Requirements

Having installed the SOAR distribution in your system. You can download SOAR 8.5.1 from:

http://sitemaker.umich.edu/soar/soar_software_downloads

To run SOAR demos, you will need also the TCL/TK. For windows, the binary with the distribution can be downloaded from:

http://prdownloads.sourceforge.net/tcl/tcl805.exe

You will need also the J2SDK 1.4.2.

5.3.5.3 Configuration

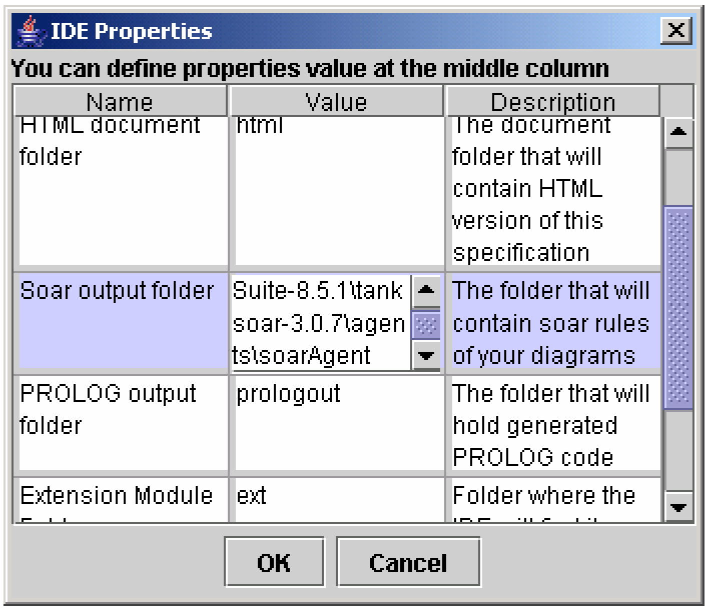
To build the SOAR module, you should type, in a command console

```
ant modsoar
```

Then, after launching the editor with

```
ant runide
```

Now it is time to setup where you want your code to be allocated. SOAR Tank demo requires that customized tanks are allocated at C:\Program Files\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents. Therefore we set the `outputFolder property to` *C:\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents\soarAgent* as shows Figure 89. The project properties dialog window is opened with the option shown in Figure 27.



Name	Value	Description
HTML document folder	html	The document folder that will contain HTML version of this specification
Soar output folder	Suite-8.5.1\tank soar-3.0.7\agen ts\soarAgent	The folder that will contain soar rules of your diagrams
PROLOG output folder	prologout	The folder that will hold generated PROLOG code
Extension Module	ext	Folder where the

Figure 89. Project properties to modify the Soar output folder

Now, you should open the demo specification file, which is the *tanksoar.xml* file, allocated in the *examples* folder. Then, selecting the *generate* option of the soar module as shows the Figure 90 to generate code

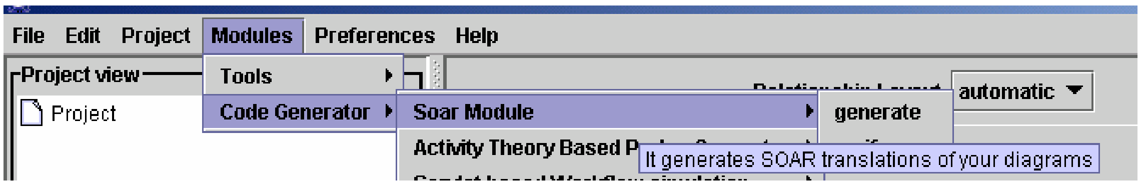
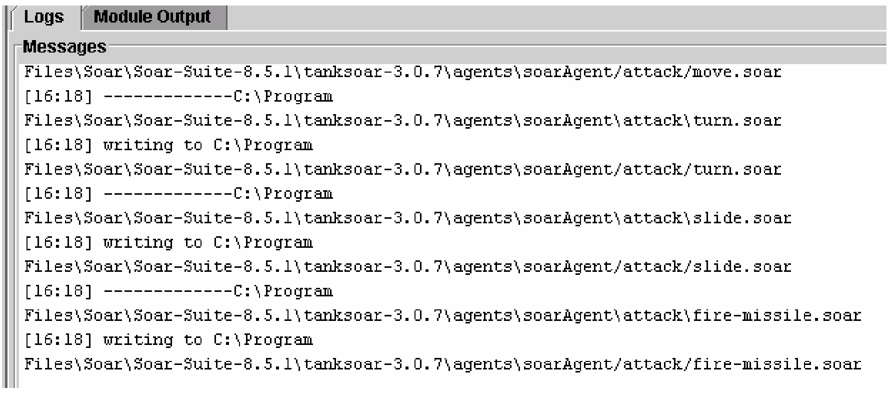


Figure 90. Code generation option for SOAR

The message panel should show something like the output of the Figure 91



```
Files\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents\soarAgent/attack/move.soar
[16:18] -------------C:\Program
Files\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents\soarAgent\attack\turn.soar
[16:18] writing to C:\Program
Files\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents\soarAgent/attack/turn.soar
[16:18] -------------C:\Program
Files\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents\soarAgent\attack\slide.soar
[16:18] writing to C:\Program
Files\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents\soarAgent/attack/slide.soar
[16:18] -------------C:\Program
Files\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents\soarAgent\attack\fire-missile.soar
[16:18] writing to C:\Program
Files\Soar\Soar-Suite-8.5.1\tanksoar-3.0.7\agents\soarAgent/attack/fire-missile.soar
```

Figure 91. Output after selecting code generation option

Now you should open SOAR by choosing Tank Soar from the Windows Start menu or starting the corresponding batch or shell file in other OS

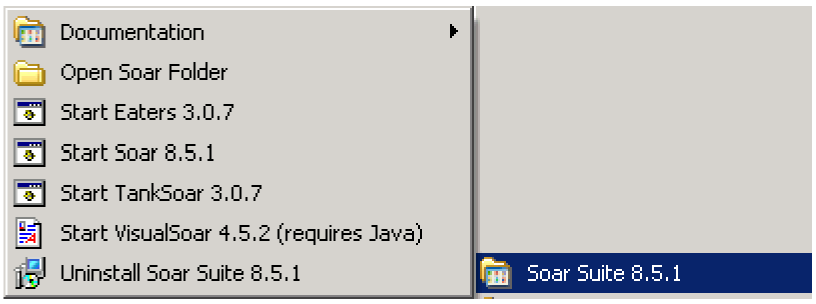


Figure 92. Starting the Tank SOAR demo

5.3.5.4 User manual

TODO

5.3.5.5 Development issues

TODO

5.3.6 Servlet based workflow simulator module

This servlet was created as a demonstration module for the Juul Bookhandle case study shown in next section. This module generates servlets that allow a simulation of the behaviour of agents in workflows. It is useful to test the dynamic behaviour of an agent organization when executing workflows.

5.3.6.1 Output of the module

This module produces a **servlet** that implements the execution workflow of tasks defined with the IDK. The module can be called from the IDK Editor or directly from the console.

5.3.6.2 Requirements

This implementation requires an application server that supports servlet execution. In this case, we used the *resin* application server (http://www.caucho.com). It is very easy to configure and launch. In any case we provide further instructions for this case study configuration.

The IDK includes the *resin* distribution and its associated licenses.

5.3.6.3 Configuration

Open the build.xml file included in the IDK distribution and change the property *resinfolder* so that it points at the home folder where the resin server is installed

<!-- Change this property to the folder where resin is installed --> <property name="resinfolder" location="resin-2.1.11"/>

To compile the module, use the following command (windows)

```
ant modservlet
```

Once it is compiled, launch the editor, load the specification file (juul.xml in this case) and:

1. Open project properties (Project-> properties), and set the property named "document folder" to the *doc/WEB-INF/classes* folder under the *resin* home folder.

2. Go to the main menu selecting *Modules -> servlet based workflow simulation -> generate.*

Figure 93. Invocation of the servlet generator

If everything goes as expected, it will start generating code of the servlet, note that the module output will show where it is writing files. If some part of the specification is not correct, it will show red messages showing the failures and what to do to solve them.

Figure 94. Confirmation message shown in the IDK Editor

The IDK distribution has a preconfigured resin application server. This server has a special file, named *resin.conf*, that is read by the application server to know what ports to use, where are the www document root, etc. To run the generated servlet, in the IDK we added the following lines to the *resin.conf* file (NO NEED TO DO THIS):

```
<web-app id='juul'>
<context-param info='An application information string'/>
<servlet servlet-name='juul' servlet-class='Juul'>
<init-param info='A servlet information string'/>
</servlet>
</web-app>
```

To start the web server, a specialised task has been defined in the build.xml included in IDK. To run this task from command line console you should type (windows & unix):

```
ant launchhttpserver
```

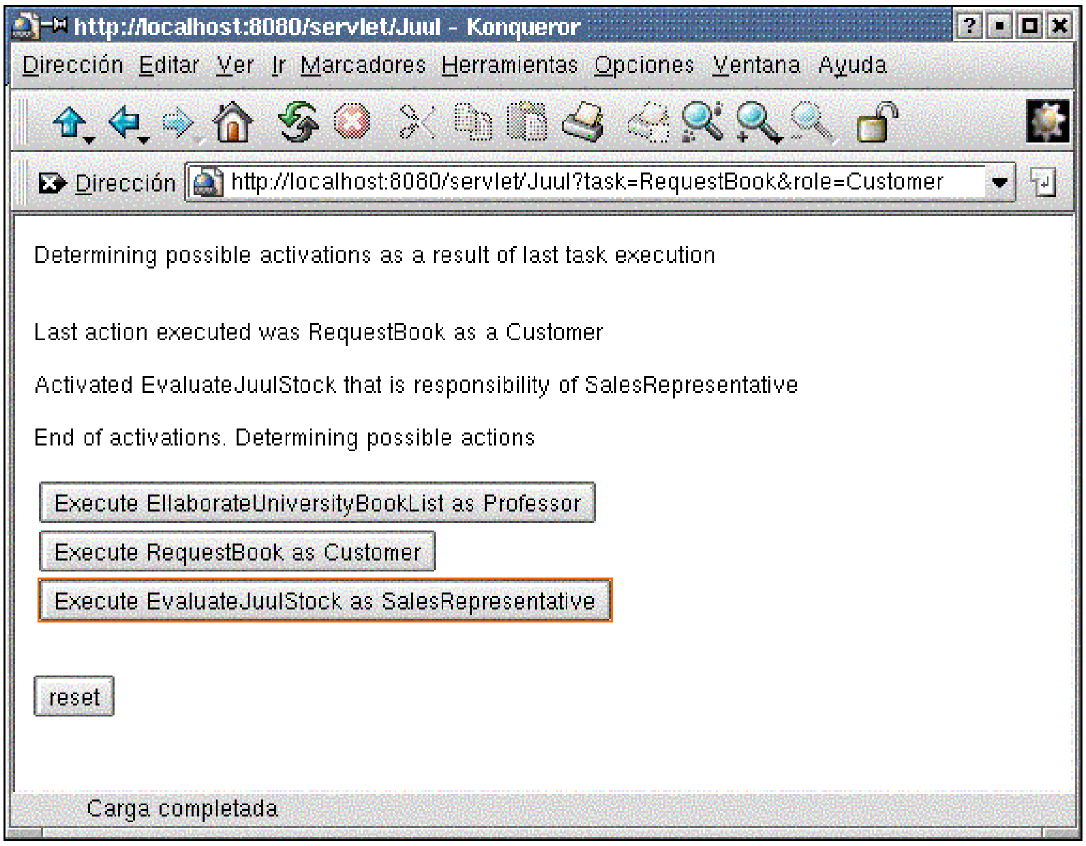
The output will be something like:

```
C\foo\>ant launchhttpserver

Buildfile: build.xml

launchhttpserver:
[java]    Resin    2.1.11    (built    Mon    Sep    8    09:36:19    PDT    2003)
[java] Copyright(c) 1998-2003 Caucho Technology. All rights reserved.

[java]   Starting   Resin   on   Wed,   18   Feb   2004   19:37:44   +0100   (CET)
[java]      [2004-02-18      19:37:46.892]      initializing      application
http://localhost:8080/
[java]      [2004-02-18      19:37:46.892]      initializing      application
http://localhost:8080/java_tut
[java]      [2004-02-18      19:37:46.892]      initializing      application
http://localhost:8080/juul
[java]      [2004-02-18      19:37:46.892]      initializing      application
http://localhost:8080/examples/tags
[java]      [2004-02-18      19:37:46.892]      initializing      application
http://localhost:8080/examples/tictactoe
[java]      [2004-02-18      19:37:46.892]      initializing      application
http://localhost:8080/examples/navigation
[java]      [2004-02-18      19:37:46.972]      initializing      application
http://localhost:8080/examples/xsl
[java]      [2004-02-18      19:37:47.363]      initializing      application
http://localhost:8080/examples/templates
[java]      [2004-02-18      19:37:47.363]      initializing      application
http://localhost:8080/examples/login
[java]           http           listening           to           *:8080
[java] srun listening to 127.0.0.1:6802
```

Once launched, the server will be accessible from this URL: http://localhost:8080/servlet/Juul

5.3.6.4 User manual

Initially, only tasks that do not get inputs from other tasks will be shown as form buttons. By pressing on one of them, the servlet will check what tasks are enacted and will add new buttons to the form. Recently enacted tasks will be highlighted with an orange frame. Besides, the form will tell you what happened when you pressed the button the last time.


http://localhost:8080/servlet/Juul - Konqueror
Dirección Editar Ver Ir Marcadores Herramientas Opciones Ventana Ayuda
Dirección http://localhost:8080/servlet/Juul?task=RequestBook&role=Customer

Determining possible activations as a result of last task execution

Last action executed was RequestBook as a Customer

Activated EvaluateJuulStock that is responsibility of SalesRepresentative

End of activations. Determining possible actions

Execute EllaborateUniversityBookList as Professor
Execute RequestBook as Customer
Execute EvaluateJuulStock as SalesRepresentative
reset
Carga completada

Figure 95. Navigator window showing selected tasks

Figure 95 shows a snapshot of the servlet. Each button represents a task that can be activated. It also has text presenting who is supposed to execute the task.

Be advised that some tasks enact other workflows. So it would not be surprising that when you thought that the workflow had finished, it turns out that the browser keeps on showing you new buttons.

5.3.6.5 Development issues

In this case, the target platform is a J2EE Servlet. This servlet simulates a workflow execution by showing the user what tasks are available on executing a concrete one. Each task is associated to a concrete role, so the user has to be informed about who is supposed to be executing each task.

This is a code generation module, so we define the new class as an inheritor of *ingenias.editor.extension.BasicCodeGeneratorImp*. Taking the template as starting point, the new module has to generate a data structure that fits into the repeat-var structure that has been defined in the previous section. To do so, the program has to traverse the diagrams.

On traversing the diagrams, we realised some problems that did not appear in the initial prototype:

- **Determine which task goes first.** The program has to infer which tasks are the first that enable other tasks. This was deduced by locating those tasks that do not take inputs from any task. As a consequence, there can be more than one initial task.
- **What if a task enacts two tasks?** In this case, the program has to take into account that there can be multiple states in a specific moment. Information flow could select any path in current structure.

The source code for the servlet generator is here. To execute it, you will need the INGENIAS distribution. This code can be executed standalone or from within the IDK.

Diagram Entity	Expected behaviour and constraints	Target Platform representation
Task	When a task is executed, it enacts other tasks in the same or other workflows. Every task must have a role associated with relationships *WFResponsible*	Form buttons Data structures in the servlet Strings
Role	The executor of a task. A role has to have at least one task assigned	Just a string
WFConnects	It is a relationship that induces the workflow structure. This relationship means that the output of a task is connected to the input of another	if-then-else to codify state transition
WFResponsible	Associates roles and tasks	Implicit

Table 1. Mapping description to generate Servlets

5.4. CONCLUSIONS ON MODULES

Modules are an open extension mechanism for the IDK. Although there are some provided with the IDK distribution, we encourage the development of new modules to cope with specific issues of particular agent development, specially when considering a new target platform or a concrete implementation architecture, which can be different to what we provide.

We have tested the complexity of module development with our students and we have realized that it took them around one month (considering that they worked just partial time on this task, as they have other assignments in their courses) to build a new module. They took as input a description of meta-models and the code of some module as an example. Therefore, we consider that the effort is quite reasonable, and their benefits great when considering the development of a real multi-agent system.

6. Case Studies

The IDK currently incorporates several cases of study. In this section we introduce briefly each one of them and what modules could be applied to them. In principle, any module can be applied to any specification. However, some of them are specially adapted to specific configurations. This is one of the reasons why we have considered that module development should be open to any user of IDK.

6.1. Collaborative Information Filtering

This case study describes how agents in an organization interact in a workflow. There are two types of agents: Personal Agents and Community Agents. Personal Agents represent the interests of individuals in the system. Their purpose is to provide useful information to their users. Community Agents, on the other side, correspond to agents willing to group together Personal Agents with common interests and keep track of the evolution of the interest of the community.

This case study was performed following the INGENIAS methodology and recommended design steps. The documentation can be also reviewed in the INGENIAS web site (http://grasia.fdi.ucm.es/ingenias)

The file containing the specification for the IDK is *colaborative filtering.xml*

6.1.1 Modules

This case study only provides the specification, but you can try to apply the following modules:

- Servlet Based Workflow simulation. Described in section 5.3.6.
- JADE module. Described in 5.3.2.
- JADE Leap module. Described in 5.3.3.

6.2. Booksellers

We chose as case study the Juul Boklander case study written by Espen Andersen from http://www.aspen.com. The case study is a bookseller company that sells books to university students. The company has an agreement with professors and students to obtain a list of books used in their courses and sell them at special prices. The bookseller is considered an organization in which there are departments in charge of sales and departments in charge of logistics. Sales can be conventional or through internet. In this case, the goal is how to define electronic sales.

This case study was performed following the INGENIAS methodology and recommended design steps. Different steps of the development process are available:

- *juul-analysis-elab.xml*
- *juul-analysis-init.xml*
- *juul-design-elab.xml*

6.2.1 Modules

This case study provides several suitable files for the following modules:

- **Servlet Based Workflow simulation.** You can use as input for this module the following files:
  - o *juul-design-elab-servlet.step1.xm*: example with failures
  - o *juul-design-elab-servlet.step2.xm*: example with failures

    - *juul-design-elab-servlet.step3.xml:* **operational example**.
- **JADE module**. You can use as input for this module the following files
    - juul-design-elab-JADE.step1.xml. example with failures
    - juul-design-elab-JADE.step2.xml. example with failures
    - juul-design-elab-JADE.step3.xml. **example operational**.
- **JADE Leap module.** You can use as input for this module the following files:
    - juul-interaction.xml. **example operational**.

6.3. QUAKE

This example was developed by: Guillermo Jimenez (gui_jim@hotmail.com). This is case study only considers modelling with the INGENIAS specification language in order to show its ability to deal with a complex environment such as the QUAKE game.

The example is based on a demo video of an implementation of quake bots made with PROLOG. The video is in Spanish and it is available at http://grasia.fdi.ucm.es/video

Current specification is on file *quake.xml.*

6.3.1 Modules

This is a modelling case study only, but you can try to apply the following modules:

- Servlet Based Workflow simulation. Described in section 5.3.6.
- JADE module. Described in 5.3.2.
- JADE Leap module. Described in 5.3.3.

6.4. ROBOCODE

This case study is not finished yet. The purpose of this case study is to translate ROBOCODE primitives to the IDK front end, and leave coding tasks apart. ROBOCODE is a game developed by IBM for educational purposes, and is available at http://robocode.alphaworks.ibm.com. In the game, you code the behaviour of a tank or a team of tanks in order to win battles.

This case study comes with an example of what would be the expected results of each stage of the INGENIAS methodology:

- *robocode-inception.xml*
- *robo-elaboration.xml*

6.4.1 Modules

- Robocode module. To be provided.

6.5. SOAR

The case study developed by Juan Antonio Recio (jareciog@ucmail.ucm.es) and consisted in adapting the Tank SOAR demo to INGENIAS. Adaptation process transformed the rule based behaviour that SOAR uses to diagrams that could be represented in INGENIAS. Basically, it expresses all a tank needs to know with two kind of diagrams: task and goal, and agent diagrams.

The specification file for soar is *tanksoar.xml*

6.5.1 Modules

- **Soar module.** This module has been described in Section 5.3.5.

7. INGENIAS

This section presents the INGENIAS methodology. INGENIAS has been developed taking MESSAGE (Caire et al., 2001) as starting point. MESSAGE proposed a notation for the specification of MAS, extending UML with agent related concepts such as agent, organization, role, goals and tasks. It also adopted the Rational Unified Process as software development process and defined activities for identification and specification of MAS components in analysis, and partially in design. INGENIAS improves MESSAGE in several aspects:

- **Integration of design views of the system.** INGENIAS links concepts of different diagrams and allows mutual references.
- **Integration of research results.** Each meta-model has been elaborated attending to current research results in different areas like coordination, reasoning, and workflow management.
- **Integration of software development lifecycle.** The coupling of the Rational Unified Process and INGENIAS is stronger now. We have defined concrete activities as well as how they should be distributed along the development life-cycle.
- **Support tools.** The support tool in MESSAGE was an editor that based in a commercial meta-case tool, named METAEDIT+. The INGENIAS Development Kit (IDK) is fully programmed in the Java language and uses open source libraries, which makes it more portable, extensible, and configurable. Apart from a graphical editor, the IDK provides a framework for the implementation of modules for verification and code generation.
- **Implementation concerns.** MESSAGE did not research how specifications could help in the implementation. The IDK framework allows to translate specification diagrams into any programming language and target platform. This facility has driven us to consider a development process where it is possible to perform rapid prototyping, and where specifications and implementations feed each other.

7.1. Introducing INGENIAS

The development of Multi-Agent Systems (MAS) brings up new issues with respect to conventional software engineering practices, as it requires the integration of different concepts from the distributed artificial intelligence field, such as autonomy, agent mental state modelling, agent interactions and organization, or the definition of goals and tasks assigned to agents in a MAS.

The purpose of INGENIAS is the definition of a methodology for the development of MAS, by integrating results from research in the area of agent technology with a well-established software development process, which in this case is the Rational Unified Process (RUP). This methodology is based on the definition of a set of meta-models that describe the elements that form a MAS from several viewpoints, and that allow to define a specification language for MAS. The specification of a MAS is structured in five viewpoints:

1. the definition, control and management of each agent mental state,
2. the agent interactions,
3. the MAS organization,
4. the environment, and
5. the tasks and objectives assigned to each agent.

The integration of this MAS specification language with engineering practices is achieved by the definition of a set of activities that guide the engineering in the analysis and design phases, with the statement of the results that have to be produced from each activity. This process is supported by a set of tools, which are generated from the meta-models specification by means of a meta-modelling processor (which is the core of the IDK). MAS modelling is facilitated by a graphical editor and verification tools. As complement to these tools, there is a generic process for parameterization and instantiation of MAS frameworks, given a concrete MAS specification. The usability of this language and associated tools, and its integration with software engineering practices have been validated in several examples from different domains, such as PC management, stock market, word-processor assistant, and specially the application to collaborative filtering information systems.

7.2. META-MODELLING

Though there may be previous interpretations of what meta-modeling is, in this document we attend to the definition provided in the Meta Object Facilities (MOF) specification (OMG, 2000). This definition states that there are several levels in the definition of a language. In fact, it defines four levels where different language grammars are defined and each level defines the grammar to be used in the next level. This process could be understood as a backwards stepwise abstraction from the information level. The process ends at the M1 level, which so far has proven to be enough to UML.

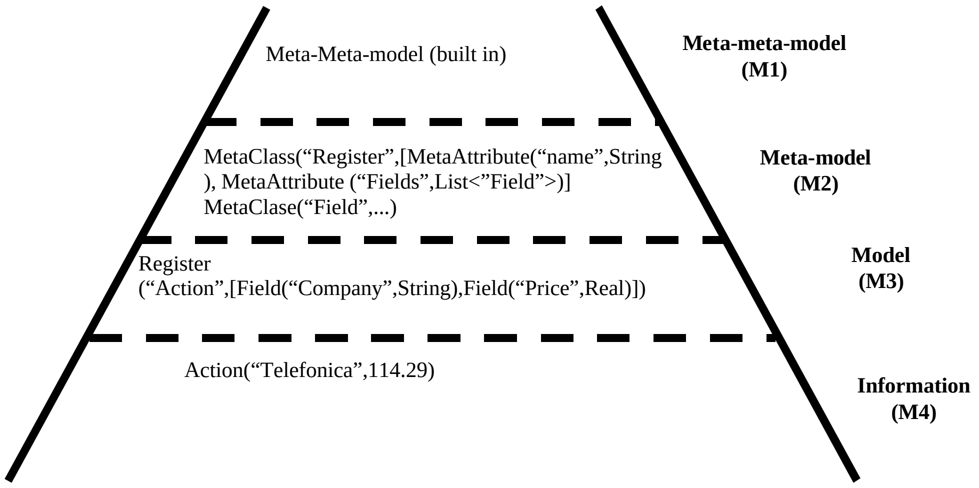
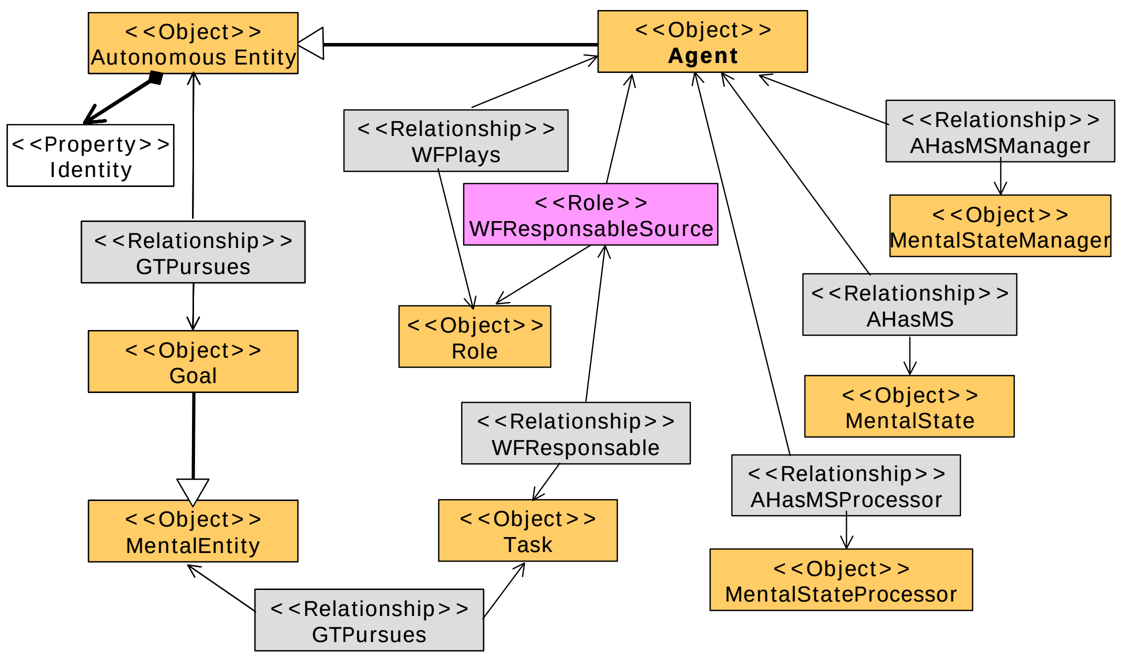

Meta-Meta-model (built in)
Meta-meta-model (M1)
MetaClass("Register",[MetaAttribute("name",String), MetaAttribute ("Fields",List<"Field">)] MetaClase("Field",...)
Meta-model (M2)
Register ("Action",[Field("Company",String),Field("Price",Real)])
Model (M3)
Action("Telefonica",114.29)
Information (M4)

Figure 96. Levels when meta-modelling according to (OMG, 2000)

In INGENIAS we use the schema of Figure 96 to structure de definition of the diagrams. However, we change the base M1 meta-meta-model and use a different one from MOF, which is the one chosen for defining UML. Instead, INGENIAS uses GOPRR (Lyytinen & Rossi, 1999) concepts which are simpler than those in MOF. In INGENIAS, after different experiences with MOF, we realized that most of the diagrams that we needed did not use most of the MOF primitives, mainly because we were not defining an object oriented language, but an agent modelling language. In this sense, we have experienced that using entity-relationship diagrams is enough for defining INGENIAS diagrams. And a suitable language to define this kind of diagrams is GOPRR. GOPRR stands for Graph Object Property Relationship and Role, since these are the elements used to define any entity-relationship diagram. Furthermore, GOPRR seems to be enough to define UML diagrams. As a proof of that, METAEDIT+, a meta-case tool distributed by METACASE, implements all UML diagrams, except UML sequence diagrams.



<<Object>> Autonomous Entity
<<Object>> Agent
<<Property>> Identity
<<Relationship>> WFPlays
<<Relationship>> AHasMSManager
<<Role>> WFResponsableSource
<<Object>> MentalStateManager
<<Relationship>> GTPursues
<<Object>> Role
<<Relationship>> AHasMS
<<Object>> Goal
<<Object>> MentalState
<<Relationship>> WFResponsable
<<Relationship>> AHasMSProcessor
<<Object>> MentalEntity
<<Object>> Task
<<Object>> MentalStateProcessor
<<Relationship>> GTPursues

Figure 97. Example of meta-model defined with GOPRR

INGENIAS meta-models are defined in the M2 level. IDK implements M2 meta-models and is used to create specifications of M3 models. Therefore, instances of the meta-models, according to Figure 98, are the concrete diagrams that the developer defines (level M3) with the IDK. There is an extra level, the M4, that is supposed to hold instances of M3 models. In INGENIAS we leave this instantiation to the developer. In our experience, models at the M3 level can be expressive enough to be used as if they were M4, but we do not. However, an M4 could be considered.

Figure 97 shows an example of a meta-model M2 which is part of the agent meta-model. It is represented using a UML class diagrams and stereotypes. GOPRR primitives appear as stereotypes of the different elements of the diagram. Basically, the diagram says that an *agent* is an *autonomous entity* that *pursues goals*. *Goals* are *mental entities* that form part of the *mental state* of the *agent*. An *agent* plays *roles* and, that way, it assumes responsibilities. An agent uses *tasks* to modify its mental state and the environment. These tasks are assigned to agents directly or through roles played. Changes in the Mental state are controlled using the *mental state manager*. This entity takes care of the consistency of the *mental state* and provides the primitives to change it. Decision procedures of the agent are built in the *mental state processor*.

7.3. INGENIAS META-MODELS

INGENIAS meta-models define five viewpoints in order to define a MAS (see Figure 98).

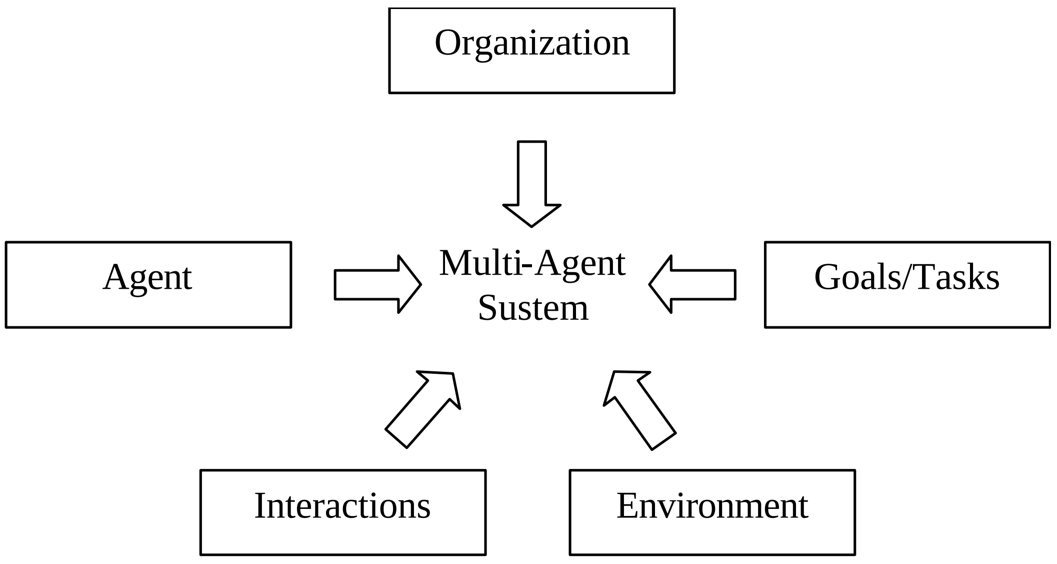


Figure 98. MAS specification viewpoints

So INGENIAS uses five meta-models that describes five types of diagrams of the same name. Entities of these meta-models, i.e. meta-entities, are not unique in the sense that anyone could be used in any of them. As a result, an entity, instance of an meta-entity, could appear in different diagrams.

- **Organization meta-model.** It defines organization diagrams. The organization is the equivalent of the MAS architecture. An organization has structure and functionality. The structure is similar to the one stated in AALAADIN framework (Ferber & Gutknecht, 1998) that later originated MADKIT. As a developer, you define the organization attending to how you think agents should be grouped. Functionality is determined when defining the goals of the organization and what workflows it should execute.
- **Environment meta-model.** It defines environment diagrams. The environment is what surrounds the MAS and what originates agent perception and action, mainly. As a developer, one of your first tasks is to identify system resources, applications, and agents. System resources are represented using TAEMS (Wagner & Horling, 2001) notation. Applications are wrappers of whatever is not an agent or a resource, and could be understood as the equivalent of objects in INGENIAS. Using these elements, a developer should be able to define how the MAS interact with the surrounding systems.
- **Task/Goal meta-model.** It describes how the mental state of agents change over the time, what is the consequence of executing a task with respect to the mental state of an agent, how to achieve goals, and what happens when a goal cannot be achieved. It also gathers dependencies among different systems or agent goals.
- **Agent meta-model.** It defines primitives to describe a single agent. It can be used to define the capabilities of an agent or its mental state. The mental state is an aggregate of mental entities that satisfy certain conditions. The initial or intermediate mental state is expressed in terms of mental entities such as those of AOP (Shoham, 1993) and BDI (Kinny, Georgeff, & Rao, 1997).
- **Interaction meta-model.** It describe two or more interacting agents. The interaction itself is a first class citizen whose behaviour is described using different languages, such as UML collaboration diagrams, GRASIA interaction diagrams, or AUML protocol diagrams. An interaction has a purpose that has to be shared or partially pursued by interaction participants. Usually it is related with some organizational goal.

An extensive detailed list of the entities, relationships, and diagrams that IDK supports, can be found in the section **¡Error! No se encuentra el origen de la referencia.**. Icons and notation appear in section **¡Error! No se encuentra el origen de la referencia.**.

7.4. FAQ ABOUT DIAGRAMS

- What is the semantics of meta-models?
    - Semantics of the meta-entities defined with the GOPRR meta-language are rather naive, nothing further from what a relationship or an entity means. However, with respect to MAS, something else could be said. In the original work of INGENIAS (Gomez-Sanz, 2002) there is an deep explanation of the intended meaning of each element, though right now it is in plain natural language. We are planning to elaborate something more formal, but there is much work to do.
- When to use them? It depends of what you intend to do. Here we provide some hints:
    - Agent diagrams.
        - When you want to express an intermediate state of the agent. Figure 99 presents an example.

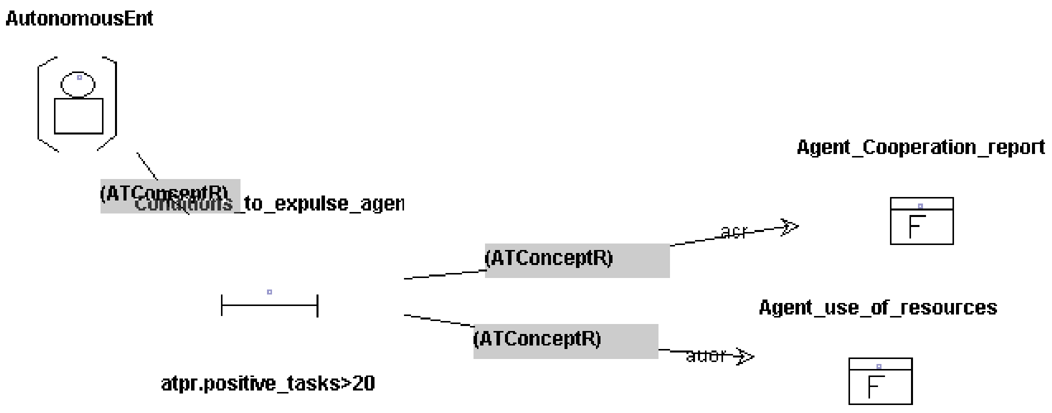


Figure 99. Mental state required to determine if an agent should be expulsed or not

        - When you need to express the initial setting of your agents
    - Interaction diagrams
        - How do I detail the interaction? You can associate different kinds of interaction specification to an interaction. Figure 100 shows an interaction that is associated with the three kinds of specification methods that the IDK supports at the moment.

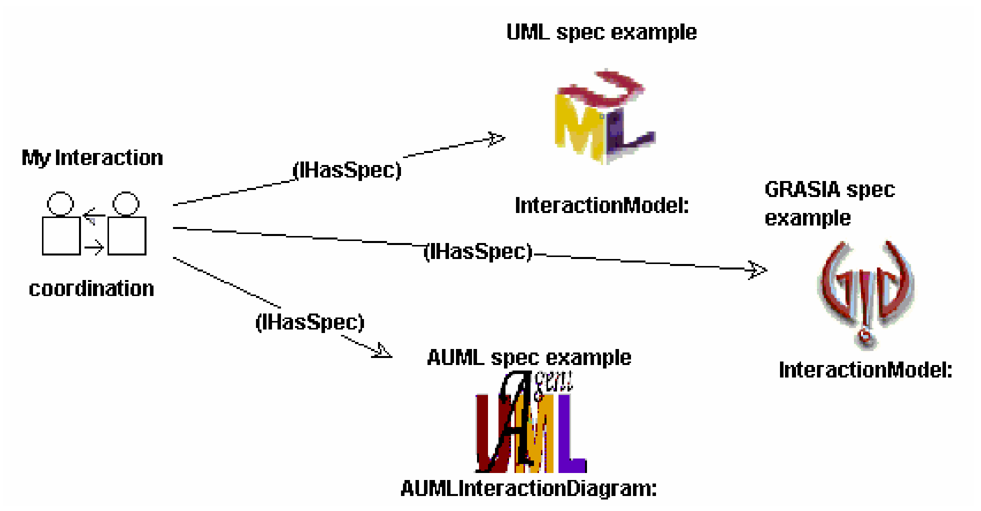


Figure 100. Interaction protocol specification by means of different specification mechanisms

8. Conclusions

The document is not finished yet and it will evolve as INGENIAS methodology does. Developing a methodology and the associated tools is not easy at all, because it is not only about theorizing what could be useful to develop a system, but a matter of experimentation.

The document presents key elements of INGENIAS Development Kit (IDK) and how to use them. We hope that this text has helped you, but in case it did not, feel free to contact us, mainly through the *ingenias.sourceforge.net* forums and mail lists. We only ask for a little patience since we have many responsibilities, however, we dedicate to INGENIAS as much time as we can.

Also, if you feel that you could contribute to INGENIAS, you can contact us at same place or by email to jpavon@sip.ucm.es or jjgomez@sip.ucm.es.

9. ANNEX I: INGENIAS METAMODEL

INGENIAS metamodel is no more included within this document. Interested readers, should visit

http://grasia.fdi.ucm.es/ingenias/metamodel

There, readers will find a MOF compliant definition of the INGENIAS concepts, relationships, and diagrams, as well as guidelines and explanations of the use of them. As an example of what readers can find, we show Figure 101 and Figure 102.

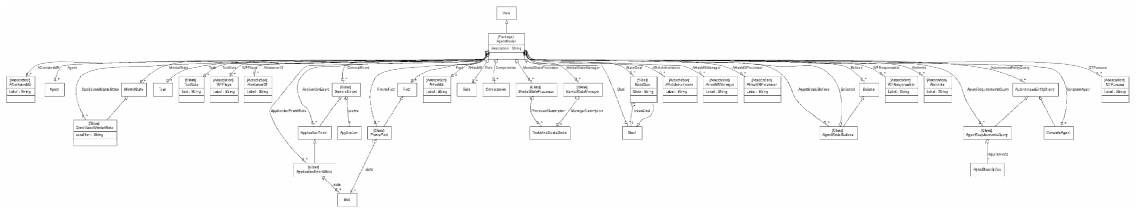


Figure 101. Meta-model for representing agent diagrams

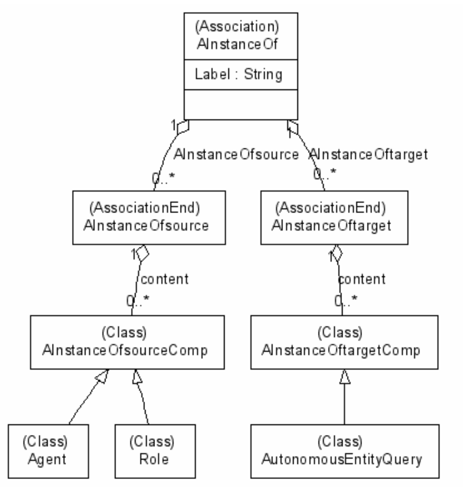


Figure 102. MOF Representation of the relationship AInstanceOf

REFERENCES

Caire, G., Leal, F., Chainho, P., Evans, R., Garijo, F., Gomez-Sanz, J. J., et al. (2001). *Agent Oriented Analysis using MESSAGE/UML*, LNCS, Vol. 2222, Springer Verlag,

Ferber, J., & Gutknecht, O. (1998, 1998). *A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems*, Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98), IEEE CS Press,

Friedman-Hill, E. (2002). Java Expert System Shell (JESS), http://herzberg.ca.sandia.gov/jess/.

Gamma, E., Helm, R., Johnson, R., & Vlissides , J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*.

Gomez-Sanz, J. J. (2002). *Modelado de Sistemas Multi-Agente.* Unpublished software engineering, Universidad Complutense de Madrid, Madrd.

Kinny, D., Georgeff, M., & Rao, A. (1997). *A Methodology and Modelling Technique for Systems of BDI Agents*. Australian Artificial Intelligence Institute.

Lyytinen, K. S., & Rossi, M. (1999, 1999). *METAEDIT+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment*, L, Vol. #1080, Springer-Verlag,

OMG. (2000). *MOF. Meta Object Facility (specification)* (No. *V.1.3.* formal).

Pavón, J., & Gómez-Sanz, J. (2003). *Agent oriented software engineering with INGENIAS.* Paper presented at the International Central and Eastern European Conference on Multi-Agent Systems, Springer Verlag.,

Shoham, Y. (1993). Agent Oriented Programming. *Artificial Intelligence, 60*, 51-92.

Wagner, T., & Horling, B. (2001, 2001). *The Struggle for Reuse and Domain Independence: Research with TAEMS, DTC and JAF*, Proceedings of the 2nd Workshop on Infrastructure for Agents, MAS, and Scalable MAS,