INGENIAS Development Kit Manual

IDK MANUAL. VERSION: 2.8

DATE: 19/3/2009

Authors: Jorge J. Gómez Sanz, Juan Pavón, Iván García-Magariño

Facultad de Informática UCM

http://grasia.fdi.ucm.es

grasia@fdi.ucm.es

This document presents the INGENIAS Development Kit (IDK), a set of tools for the specification, implementation, testing and validation of multi-agent systems (MASs). It is oriented to researchers looking for an agent-oriented specification tool and software 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. This functionality can be extended with modules that perform automatic code generation, testing and analysis of the specifications. Some of these modules are already provided with the IDK distribution, but developers can also create new modules for a particular application (i.e., 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 for MASs, as defined in the INGENIAS project.

INDEX

1. INTRODUCTION

This document introduces the *INGENIAS Development Kit* (***IDK***), a set of tools for the development of multi-agent systems (MAS). INGENIAS follows a model-driven engineering approach, in the sense that developers create and work with graphical models of multi-agent systems. These models can be transformed, by using IDK, into implementations for their deployment and execution. IDK promotes also agile application development, by supporting quick iterations of specification-implementation-testing cycles.

IDK is structured in the following tools:

- *A **Graphical Editor***. This allows the developer to create and modify specifications of a MAS using agent concepts. The underlying agent model is defined in the INGENIAS methodology, but it can incorporate standards such as FIPA and AUML.
- ***Modules***. They allow working with MAS specifications to perform verification of MAS 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 agent 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 developers 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 introduces the use of the IDK, and can be used as a user's manual for the IDK. Some sections provide also reference information, more concretely, a specification of the meta-models that are supported by the IDK. 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 UCM-*grasia!* research group (http://grasia.fdi.ucm.es), and should be referenced as follows:

> [Gómez-Sanz and Pavón, 2004] Gómez-Sanz, Jorge and Pavón, Juan (2004). *INGENIAS Development Kit (IDK) Manual*. Facultad de Informática, Universidad Complutense de Madrid, Spain. Available at http://ingenias.sourceforge.net

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

1.2. Credits

To the members of the Grasia! Research group, especially to the authors of this technical report (Jorge J. Gómez Sanz, Juan Pavón, and Iván García Magariño). Also to the rest of the people that collaborated in the development of the IDK (Ruben Fuentes), and those who participated as beta-testers (Guillermo Jiménez, Juan Antonio Recio, Carlos Celorrio, Alberto Fernández, and many others). Also to collaborators from other institutions, who have tested the IDK and provided new modules, especially from the Univ. Murcia (Juan A. Botía) and Univ. Vigo (Juan Carlos González Moreno).

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

1.3. Guide to read the document

The document is structured in three parts.

The first part is oriented to almost any user, and describes basic use of IDK for modelling MAS: how to install the IDK (chapter 2), how to use the editor (chapter 3), and the automatic backup functionality (chapter 4).

The second part is related with IDK modules and describes how to generate an implementation of a MAS from its specification. Chapter 5 describes what is an IDK module, how to use it, and how to create new modules to extend the functionality of the IDK, for instance, to build a customized code generator. Chapter 6 describes the basis of INGENIAS, and more concretely of the INGENIAS meta-model, which is required to build new IDK modules.

The third part (chapter 7) presents a complete example on how to use IDK and the INGENIAS software process. This chapter is recommended as a guide to the development of a MAS using IDK, from specification to implementation.

Finally, the references section provides a list of documents that can be useful to complement the information on INGENIAS and the IDK.

2. Getting started with the IDK

This section describes the requirements for using IDK (section 2.1), how to install the IDK (sections 2.2 and 2.3), and a simple example of a system with two configurations (one with a single agent, and another with several agents). This example, in sections 2.5. and 2.6, shows how to specify a MAS, how to generate automatically an implementation, and how to deploy the agents on a JADE agent platform [15].

The IDK is officially distributed from http://ingenias.sourceforge.net. There are two possibilities to download and install the IDK. The first option consists in downloading a binary with the IDK installer (currently available for three platforms: MacOS, Linux, and Windows), and it is described in section 2.2. The second option consists of downloading a zip file and decompressing it at the appropriate location, as described in section 2.3.

2.1. Requirements for using IDK

IDK is implemented on Java, so it should work on any platform supporting Java. However, it is recommended J2SDK **1.6.0** or higher. Lower versions do not have some features such as regular expressions handling or clipboard transfer utilities. Also, we have observed some GUI misbehaviours if lower versions are used.

JDK can be downloaded from: http://java.sun.com/javase/downloads/index.jsp

IDK also uses Apache Ant build tool. This is a tool for making repetitive tasks, for instance, during the compilation and build phases. Configuration is specified on a XML file, so it can be applied in almost any platform. For source code generation, IDK produces an ant configuration file, with facilities to compile and run the generated system. This tool and its installation instructions are available at: http://ant.apache.org.

Optionally, the Eclipse tool (www.eclipse.org) can be used. This platform can assist users in running the "ant" commands. Moreover, an experimental IDK plugin is currently being developed for the Eclipse tool. The Eclipse tool and the necessary installation instructions are available from its website.

2.2. Installing the IDK with Installer

The easiest way to install the IDK is with the installer, which is provided for Windows, Linux and MacOS platforms.

In Windows, download the installation file and double click on it.

In Linux, execute the installer with the three following steps:

1. Download the Installation File.
2. Right click on the downloaded file, and extract the file.
3. Left click on the extracted file to start the installer.

When starting the installer, the first thing to do is the selection of a language for the installation, as Figure 1 shows.

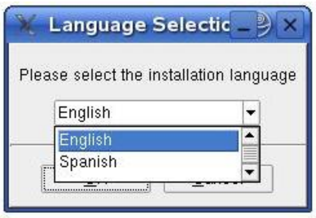
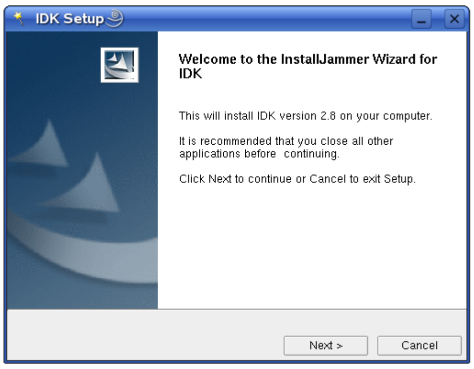


Figure 1: Selection of the Language

The next window (see Figure 2) shows a welcome and request for a confirmation to install IDK. It is recommended to close all the other applications before continuing. After closing the others applications, click the “Next” button to continue or “Cancel” to stop the installation.



Welcome to the InstallJammer Wizard for IDK

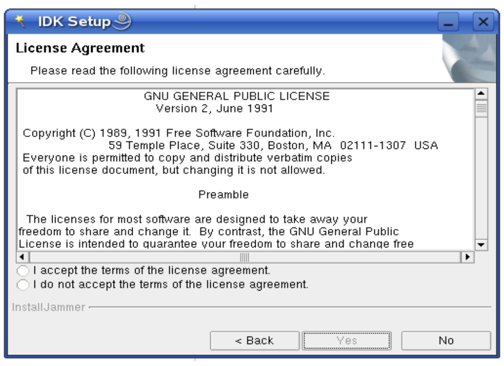
This will install IDK version 2.8 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue or Cancel to exit Setup.

Figure 2: Welcome Screen

The next step (see Figure 3) relates to the License. IDK is distributed with a GNU General Public License. After reading it, indicate whether the terms of the license agreement are accepted. This is mandatory to proceed with the installation, by clicking in the button labelled with “I accept the terms of the license agreement” and, after this, clicking the “Yes” button.



License Agreement

Please read the following license agreement carefully.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

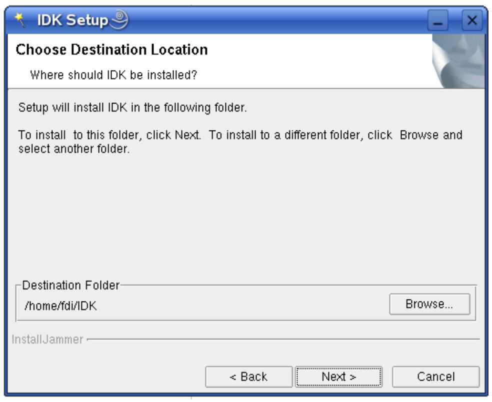
The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free

I accept the terms of the license agreement.
I do not accept the terms of the license agreement.

Figure 3: License Agreement

The installer checks out if the adequate Java Version is installed. If the installer does not find a right Java version, it asks the user where it is installed. If it cannot find the appropriate Java version, the installer tells the user where Java 6 can be downloaded, and it stops the execution of the installation.

Then, the installer proposes a default location to place the IDK application. It is possible to select another installation path, by pressing the “Browse” button (see Figure 4). Currently, no other configuration parameter is required for installing IDK. Then click “Next” to proceed with the installation.



Choose Destination Location

Where should IDK be installed?

Setup will install IDK in the following folder.

To install to this folder, click Next. To install to a different folder, click Browse and select another folder.

Destination Folder
/home/fdi/IDK

Figure 4: Choose Destination Location

The installation (see Figure 5) will copy the files to the corresponding path location. The user is informed of the progress with a blue bar.

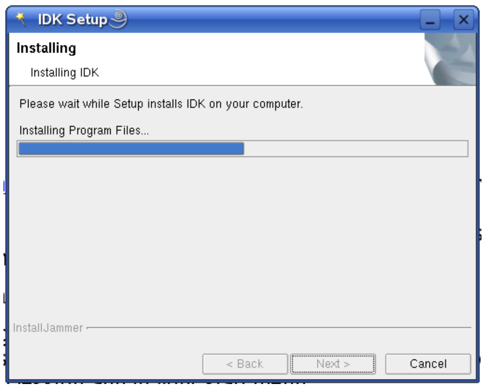

IDK Setup
Installing
Installing IDK
Please wait while Setup installs IDK on your computer.
Installing Program Files...
InstallJammer
< Back
Next >
Cancel

Figure 5: Progress of the IDK Installation

The last window (see Figure 6) should indicate that the IDK has been successfully installed. There are three options at the end: to read a file indicating some release notes; to launch the IDK; and to create a desktop shortcut for IDK. Once the desired options are checked, press the “Finish” button to finish the installation.

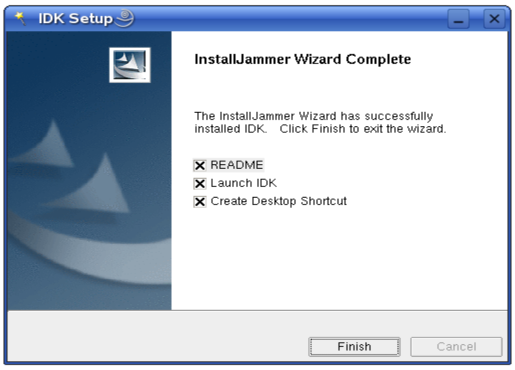

IDK Setup
InstallJammer Wizard Complete
The InstallJammer Wizard has successfully installed IDK. Click Finish to exit the wizard.
README
Launch IDK
Create Desktop Shortcut
Finish
Cancel

Figure 6: Successful Installation Screen

Depending on the options selected in the last step, the IDK is launched and the README file is presented to the user. To test the installation click the desktop shortcut (see Figure 7) or select the application in the corresponding menu (see Figure 8).

Figure 7: IDK Icon

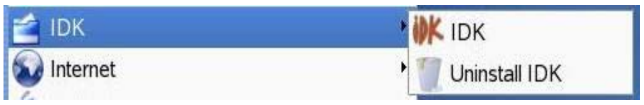


Figure 8: IDK in the Programs Menu

2.3. Installing the IDK from Source

A more basic way to install IDK is to download and decompress a zip file in the preferred folder, for instance, c:\foo or /home/localuser/foo. One way to test the IDK is to start the editor, the main component of the IDK. From the installation directory, e.g., the *foo* directory, it can be run directly with:

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

However, it is recommend to use the *Ant* tool instead (see section 2.1 on how to install Ant). 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.

By using Ant, the IDK is started with the command:

```
foo> ant runide
```

or just:

```
foo> ant
```

Where *ant* is the script that launches the Ant system. This requires having Ant installed, and the *ant.bat* or *ant.sh* script in the *PATH* environment variable.

2.4. Installing the IDK Plugin for Eclipse

The IDK plugin for Eclipse is experimental, and it is under development. This plugin intends to facilitate integration of IDK with the Eclipse environment to manage the specification of the MAS, generate the code, an program agent components all in Eclipse.

The installation process consists of the following steps:

1. Download the zip file with the plugin to the folder that contains the "eclipse" folder of the Eclipse tool.
2. Unzip the zip file
3. Restart Eclipse.

If the IDK plugin has been successfully installed, new buttons with the IDK symbol or similar should appear in the top area, (see Figure 10).

2.5.The Hello World Example

The "Hello World" example is the typical example to introduce a new tool, so readers are familiar with it. For this reason, this manual uses it to introduce the IDK.

Initially, this implies a very simple multi-agent system, with a single agent that says "Hello World" to the outside world. In section 2.6 this example will be extended to consider more than one agent. But first we will see the basics of working with IDK.

2.5.1 Creating an IDK Project

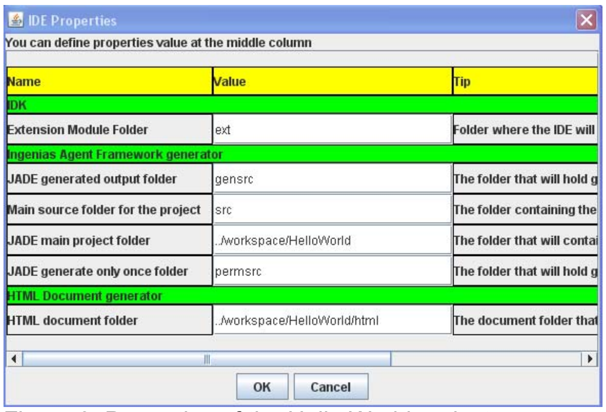
In order to use all facilities of IDK (from the graphical specification of a MAS to its implementation), the first thing to do is to create a new project. This project will have a configuration of folders to manage the MAS specifications and the different implementation files that are generated by the IDK code generation plugins. There are two ways to create the project folder structure, by using Ant facilities for IDK or by using Eclipse with IDK. The first is described below, and the use of Eclipse is described in section 2.5.2.

For creating the Hello World example project, using a shell or command window, get to the IDK folder, then to the "iaf" folder, and run:

```
iaf> ant -DmainP=../workspace/HelloWorld create
```

This will create a project named *HelloWorld* in the “IDK/workspace” folder[1]. Inside this project an empty specification can be found in “*spec*” folder. The specification is already prepared for correctly allocating the generated code.

The specification properties define parameters for code generation. Open the new specification and go to the menu “Project -> Properties” (see Figure 9). The *JADE main project folder* property is filled with the folder of the workspace that contains the MAS, and this path is relative to the *workspace* folder within the IDK installation folder. When creating a new project, this property is correctly initialised. However, if the user decides to move or rename the folder, then the user should change this property so the code generation can continue working. The path of other properties are relative to the path of the *JADE main project folder* property. For instance, the *JADE generated output folder* property specifies the path where the programming code is generated. In this folder, the code is overwritten; hence this manual recommends to upload the changes to the specification in case of writing a piece of code there. The *JADE generated only once* property specifies the path where the code is generated but not overwritten. Finally, the *Main source folder for the project* property indicates the path where the programmer can add all the necessary external programming code. These properties are initialised respectively with the *gensrc, permsrc* and *src* values by default. However, these values can be changed for specific needs. Finally, the *HTML document folder* property indicates where the HTML documentation is generated; and the *Extension Module folder* property indicates the folder from which the IDK loads extension modules.


IDE Properties

You can define properties value at the middle column

Name	Value	Tip
IDK		
Extension Module Folder	ext	Folder where the IDE will
Ingenias Agent Framework generator		
JADE generated output folder	gensrc	The folder that will hold g
Main source folder for the project	src	The folder containing the
JADE main project folder	../workspace/HelloWorld	The folder that will contai
JADE generate only once folder	permsrc	The folder that will hold g
HTML Document generator		
HTML document folder	../workspace/HelloWorld/html	The document folder that

Figure 9: Properties of the Hello World project

2.5.2 An IDK project with Eclipse

It is also possible to use IDK with Eclipse, so editing final code and working with MAS specifications can be done at the same time.

This requires the installation of the Eclipse plug-in for IDK (see section 2.4).

A new project can be created with the Eclipse plug-in for IDK, by the following steps:

1. Open the Eclipse Plugin for IDK.
2. Right button on the left area, and press “New->Other->SampleWizards->IDK Project”, as shown in Figure 10.
3. Type the name of the project (e.g., “HelloWorld”).

[1]The IDK distribution creates a workspace folder in the IDK installation folder. Note that this workspace is not the same as the default Eclipse workspace folder.

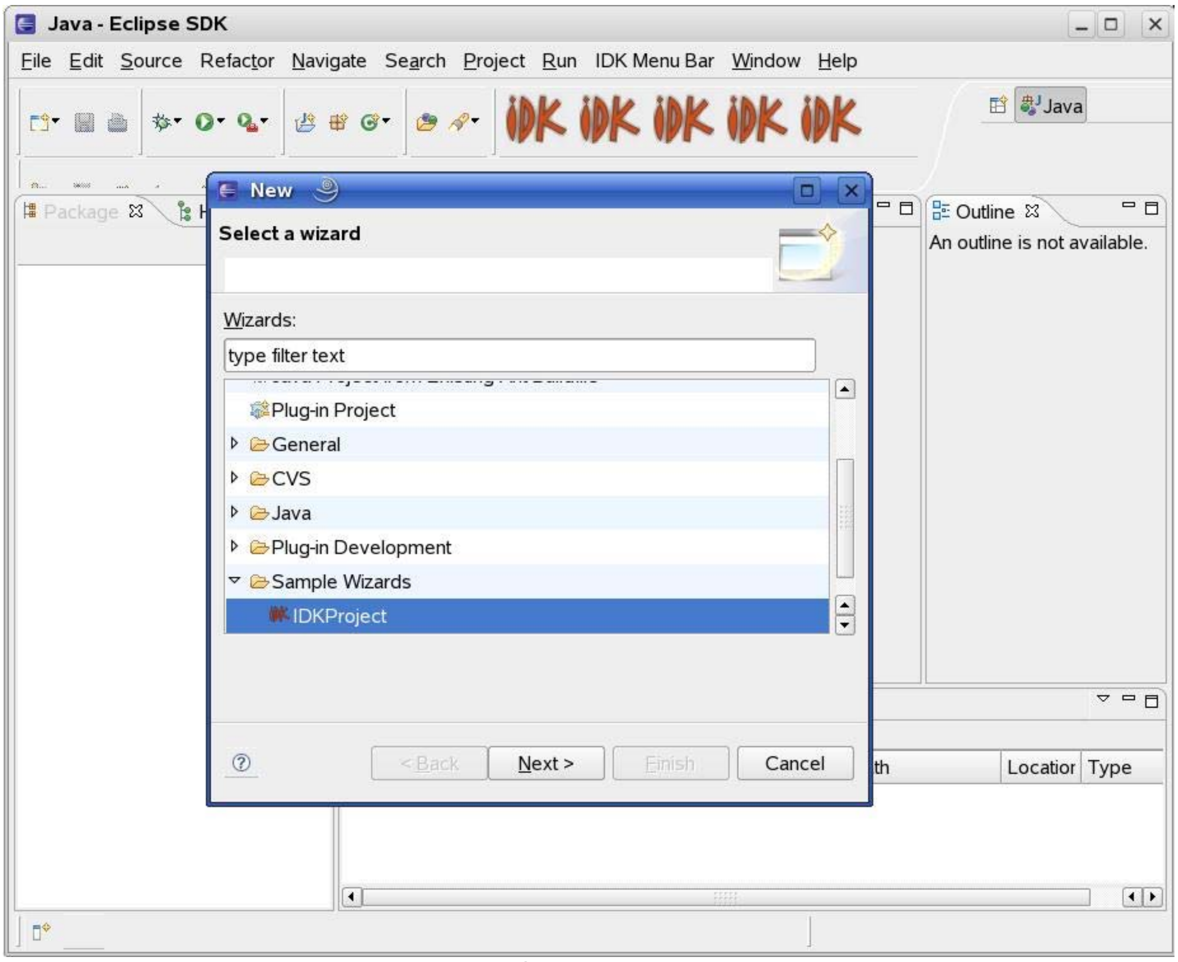


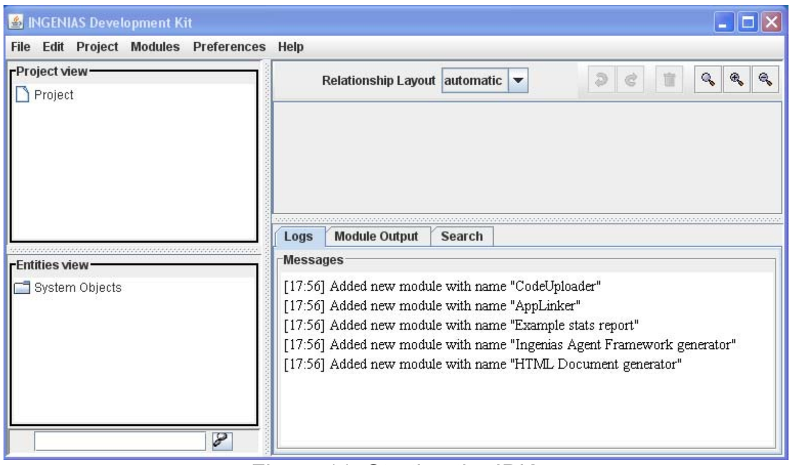
Figure 10: Creating the IDK Project

2.5.3 Starting the IDK

The IDK can be started by clicking on the IDK icon on the desktop or by selecting the appropriate menu item in the list of programs (e.g., in Windows: "Start"→"Programs" → "IDK" → IDK).

Another possibility is to run the "ant" command on the "IDK/editor" folder.

Figure 11 shows the IDK when it starts.



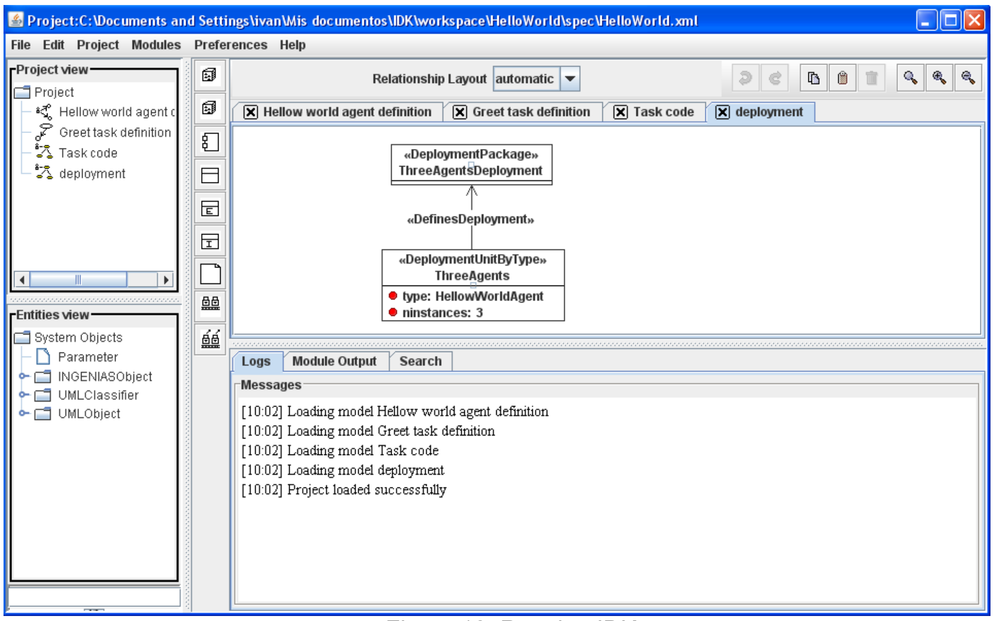
```
[17:56] Added new module with name "CodeUploader"
[17:56] Added new module with name "AppLinker"
[17:56] Added new module with name "Example stats report"
[17:56] Added new module with name "Ingenias Agent Framework generator"
[17:56] Added new module with name "HTML Document generator"
```

Figure 11: Starting the IDK

2.5.4 Opening the specification of a MAS with IDK editor

At this moment, it is possible to create the HelloWorld specification from scratch, or to open the HelloWorld specification example provided at the training pages for INGENIAS. This example can be downloaded from http://grasia.fdi.ucm.es in “Training” → ”Full Development Examples” → ”Hello World”. Then unzip the file in the “IDK/workspace” folder. To start working with the specification for the Hello World example, select the command “File → Load”, and open the following file: “IDK\workspace\HelloWorld\spec\HelloWorld.xml” (see Figure 12).

To develop this example from scratch, open the IDK\workspace\HelloWorld\spec\specification.xml” file of the recently created project. It is possible to rename the specification.xml file to “HelloWorld.xml”.


Project:C:\Documents and Settings\ivan\Mis documentos\IDK\workspace\HelloWorld\spec\HelloWorld.xml

Relationship Layout: automatic

```
[10:02] Loading model Hellow world agent definition
[10:02] Loading model Greet task definition
[10:02] Loading model Task code
[10:02] Loading model deployment
[10:02] Project loaded successfully
```

Figure 12: Running IDK

2.5.5 Working with a MAS specification

A MAS specification consists of several diagrams. These can be reviewed, created and modified with the IDK editor.

For the Hello World agent example, the reader can learn the very basic notation in the legend that is shown in Figure 13.

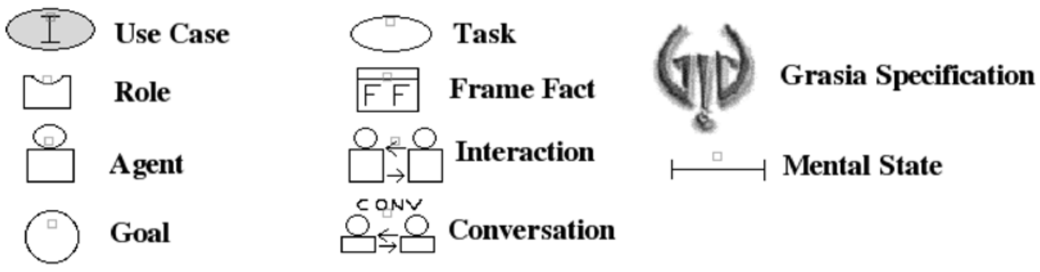


Figure 13: IDK Notation

The first diagram for this example will describe the agent that will say “Hello World”. This is created with the option “Add Agent Model” in the context menu of the “Project” item in the “Project view” (the context menu appears when clicking the right button of the mouse on the “Project” item). The agent diagrams assist designers in specifying their agents. The use cases should be defined before this agent, but this step is skipped here for the sake of simplicity.

The agent diagram in this example will describe an agent that has one goal (greet a user), and is able to perform a task (to greet), and has a mental state to represent its knowledge about the world. As one can observe, in this step agents are associated with the goals they pursue and the tasks that they are responsible for. In case of existing interactions, agents should play roles; but this is not necessary in this simple example. The agent specification of this example is drawn in several steps:

1. Insert an agent called “HelloWorldAgent” in the diagram. Agents are the basic units for building MASs. An agent is created by selecting the first icon in the column of icons at the left of the agent diagram window. Or in that window, click the right button of the mouse and select the option “Insert Agent” in the contextual menu. By clicking twice on the agent icon it is possible to change its name. It is worth noting that other specifications focuses on the definition of roles and, then, some agents are associated with these roles.
2. With a similar procedure, define a goal for the agent, and name it “Greet user”. Goals are the motor of *Belief Desire and Intentions* (BDI) architectures of MASs. Agents perform tasks and interact between each other in order to satisfy their goals. Therefore, goals keep MASs active and are necessary in every IDK specification. In this simple example, the goal of the “HelloWorldAgent” agent is to say “hello” to the user (i.e. greet the user).
3. Create a task and name the task “Greet”. Tasks are another important part of MASs, because tasks specify the actions agent can perform.
4. Create a mental state with a frame fact to represent explicit knowledge. They can be named “Initial Mental State” and “Greetings Containment” respectively. The IDK provides a mechanism to specify the mental states of agents by means of pieces of knowledge, which are denoted as *frame facts*. In this case, the “Greetings Containment” frame fact contains the sentence to greet the user.
5. Afterwards, it is necessary to link the agent called “HelloWorldAgent” with its goal, task and initial mental state. The initial mental state is related with the frame fact. For this purpose, link the elements as one can observe in Figure 14. Relationships are drawn by moving the cursor to the agent and when the shape of the cursor changes to a hand, press the mouse button and keep it pressed while moving the arrow to the element to associate.

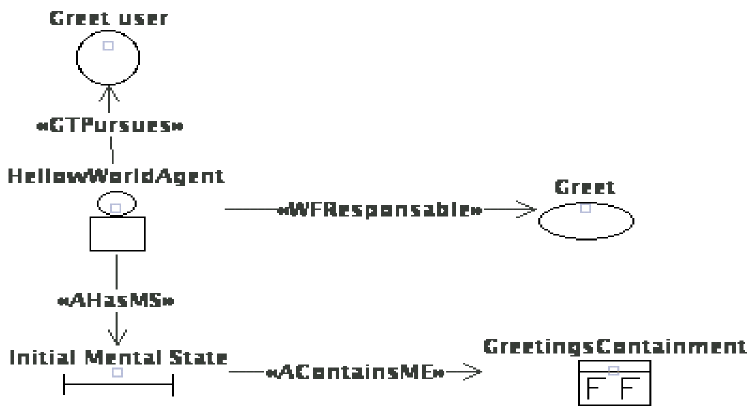


Figure 14: Agent Diagram of the HelloWorld example

It is worth noting that each relationship is stereotyped in order to show the kind of relationship between the entities. This can be changed and customized, as it is explained in section 3.2.

Once the agent is defined, the next step is the specification of the tasks that the agent can perform. A task is characterized by the goal the task intends to satisfy, and the input and output of the task. The task is executed if the agent pursues the goal the task intends to satisfy, and if the input is available in the mental state of the agent. In this case, the “Greet” task is linked with the “Greet user” goal and consumes the “Greeting Containment” frame fact. The relationship “consumes” points to the input of the task, in order to indicate that Figure 15.

It is possible to associate a task with code in Java or other programming language. This is done by associating the task with a Code Component, in which the source code is inserted, as shown in Figure 16.

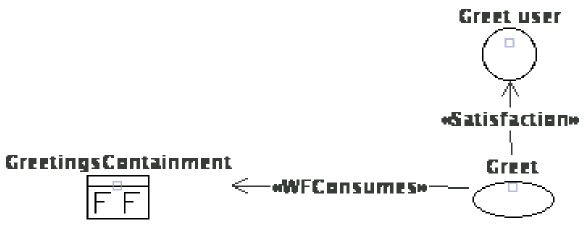
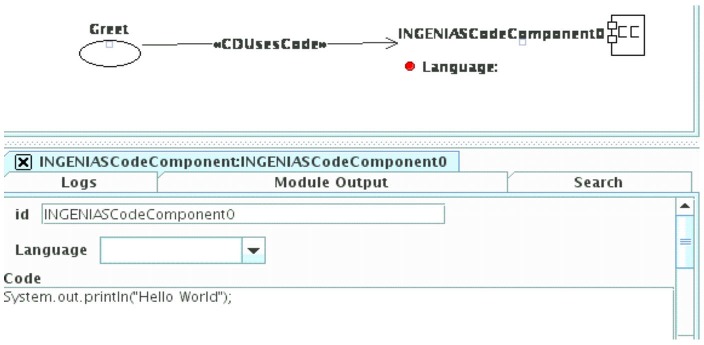


Figure 15: Definition of the Greet Task



INGENIASCodeComponent0

```
System.out.println("Hello World");
```

Figure 16: Definition of the Code Component of the Greet Task

This example is one of the simplest possibilities offered by the IDK. Nevertheless, more diagrams are necessary when developing other MASs. To begin with, the use case diagrams allow system analysts to define use cases and relate them with some goals and agent roles.

Furthermore, the interaction diagrams are arguably the cornerstone of most MASs developed with IDK. These kinds of diagrams define the social activity among agents, which makes emergent behaviours possible, such as cooperation and collective decision-making.

Moreover, organization diagrams provide a social structure which allows designers to organize MASs in a high level of abstraction.

Finally, some deployment diagrams allow users to indicate the number and types of agents that are initialised, as well as their mental states at the beginning of execution. These diagrams can also specify batteries of test, which are crucial in agile development processes and useful in every development process. Examples of the all aforementioned kinds of diagrams are included in the remaining of this manual.

2.5.6 Code generation

INGENIAS promotes a model-driven approach, which means that the code is generated from the specification (the model of the MAS that has been edited with IDK).

Code generation for the example by selecting the menu option “Modules → Code Generator → Ingenias Agent Framework generator “ and then the option “Generate” (see Figure 17). This activates an IDK module for code generation on the JADE agent platform. The activity of the code generation is shown in a Logs window of IDK.

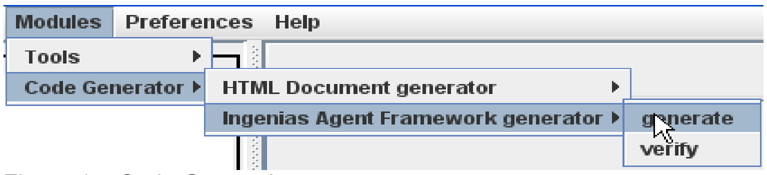
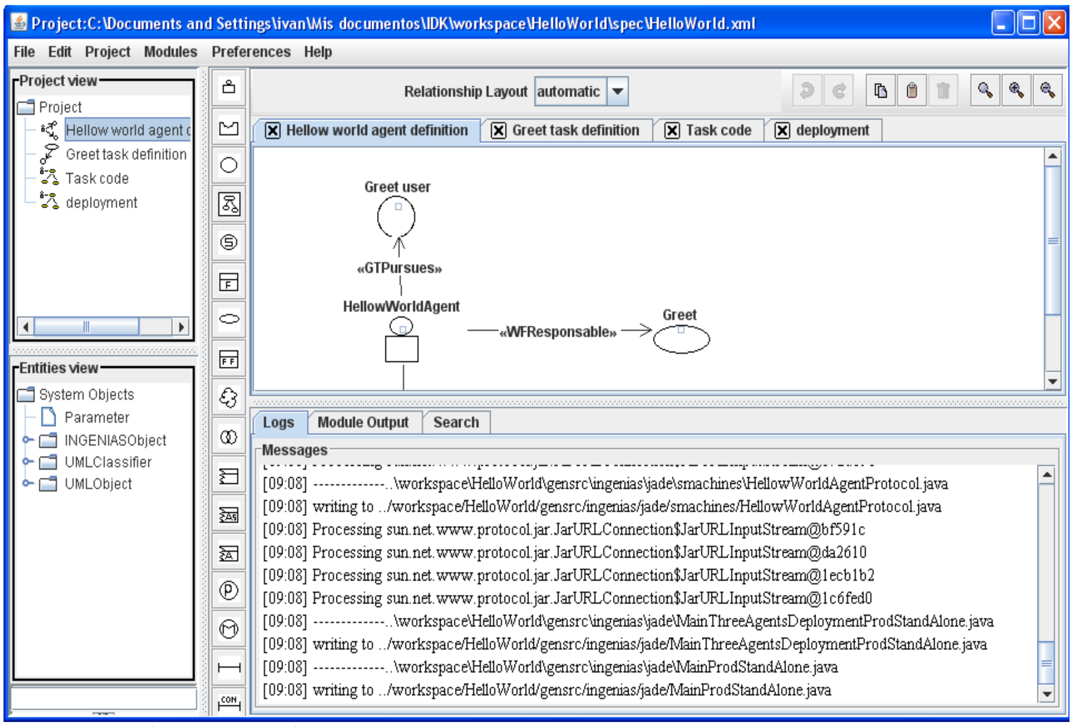


Figure 17: Code Generation

At the end, a message with a nice sound indicates that the code generation has been successful, as shown in Figure 18. The messages "writing to ..." indicates the location where the code is being generated. If there is no red "Error:..." messages and the "writing to ..." messages stop appearing, the code is successfully and completely generated.



```
[09:08] -------------..\workspace\HelloWorld\gensrc\ingenias\jade\smachines\HellowWorldAgentProtocol.java
[09:08] writing to ../workspace/HelloWorld/gensrc/ingenias/jade/smachines/HellowWorldAgentProtocol.java
[09:08] Processing sun.net.www.protocol.jar.JarURLConnection$JarURLInputStream@bf591c
[09:08] Processing sun.net.www.protocol.jar.JarURLConnection$JarURLInputStream@da2610
[09:08] Processing sun.net.www.protocol.jar.JarURLConnection$JarURLInputStream@1ecb1b2
[09:08] Processing sun.net.www.protocol.jar.JarURLConnection$JarURLInputStream@1c6fed0
[09:08] -------------..\workspace\HelloWorld\gensrc\ingenias\jade\MainThreeAgentsDeploymentProdStandAlone.java
[09:08] writing to ../workspace/HelloWorld/gensrc/ingenias/jade/MainThreeAgentsDeploymentProdStandAlone.java
[09:08] -------------..\workspace\HelloWorld\gensrc\ingenias\jade\MainProdStandAlone.java
[09:08] writing to ../workspace/HelloWorld/gensrc/ingenias/jade/MainProdStandAlone.java
```

Figure 18: Log for a successful code generation

2.5.7 Compiling and executing the code with Ant

Once source code for the project has been generated, it has to be compiled. This can be done with the Ant utility or directly from Eclipse (if the project was created with Eclipse). The first case is described here and the second in section 2.5.8.

For compiling and executing the project, open a console (in Linux, Windows and Mac). In the "IDK/workspace/HelloWorld" folder, execute the following command:

ant runProdStandAlone

This compiles the project and, in case of compiling errors, these errors are printed out in the console. If the compilation finishes successfully, then the MAS starts running. The "HelloWorld" messages will be printed out in the console. This mode of execution is called "Product Stand Alone".

The MAS can alternatively be executed in debugging mode. In this case, the user has a complete control of the execution and can watch all the execution aspects: running agents, their mental states, the interactions among agents, the events created by the external applications, the execution of tasks, and so on. For this execution mode, the user opens two consoles, both in the "IDK/workspace/HelloWorld" folder. Then, in the first console, introduce the following command:

```
> ant runjade
```

This command will start the JADE platform for agents. Then, in the other console, introduce the following command:

```
> ant run
```

This command will start the IDK debugging engine and the MAS.

2.5.8 Compiling and executing the code with Eclipse

For the execution of the multi-agent system, go through the following steps:

- Create an Eclipse project from the following source folder:
  - IDK/workspace/helloworld
- Compile and Execute the MAS by the following steps:
  - right-click on “build.xml” file, and pressing on “run as->ant build...” (see Figure 19)
  - Select “RunProdStandAlone” target and press on “run” button (see Figure 20).

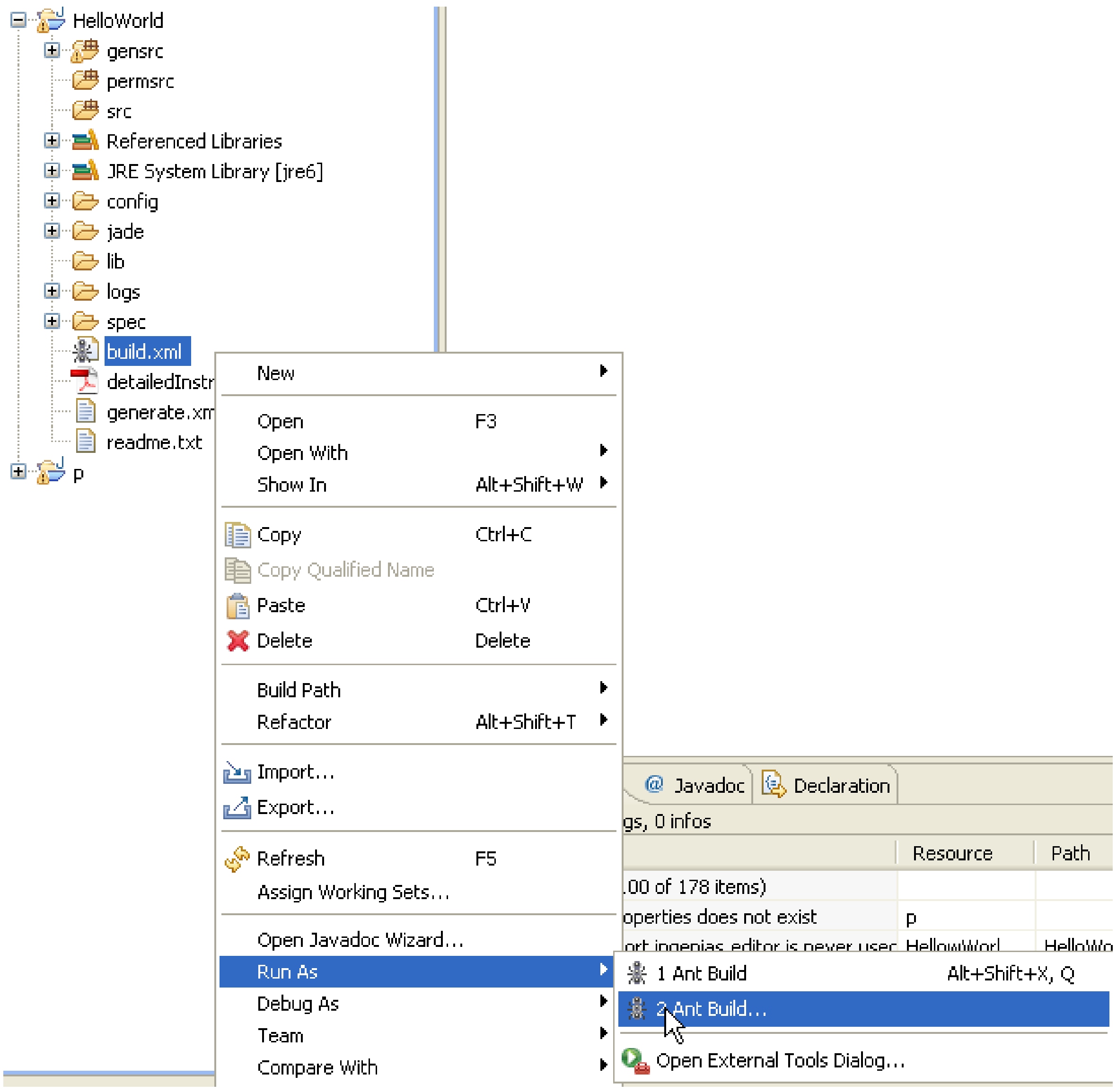
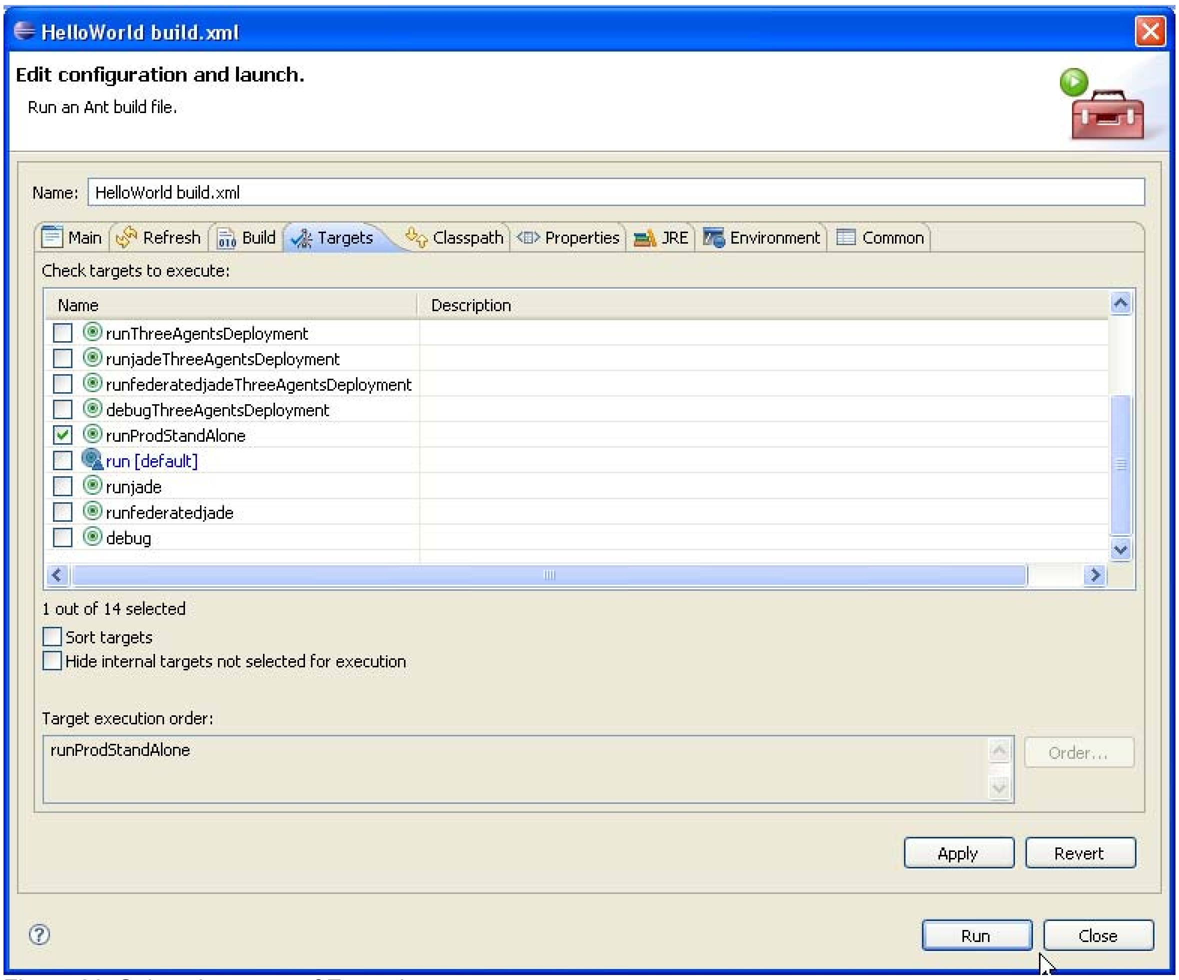


Figure 19: Compiling and Executing the Hello World Example from Eclipse



HelloWorld build.xml

Edit configuration and launch.

Run an Ant build file.

Name: HelloWorld build.xml

Check targets to execute:

	Name	Description
☐	runThreeAgentsDeployment	
☐	runjadeThreeAgentsDeployment	
☐	runfederatedjadeThreeAgentsDeployment	
☐	debugThreeAgentsDeployment	
✓	runProdStandAlone	
☐	run [default]	
☐	runjade	
☐	runfederatedjade	
☐	debug	

1 out of 14 selected

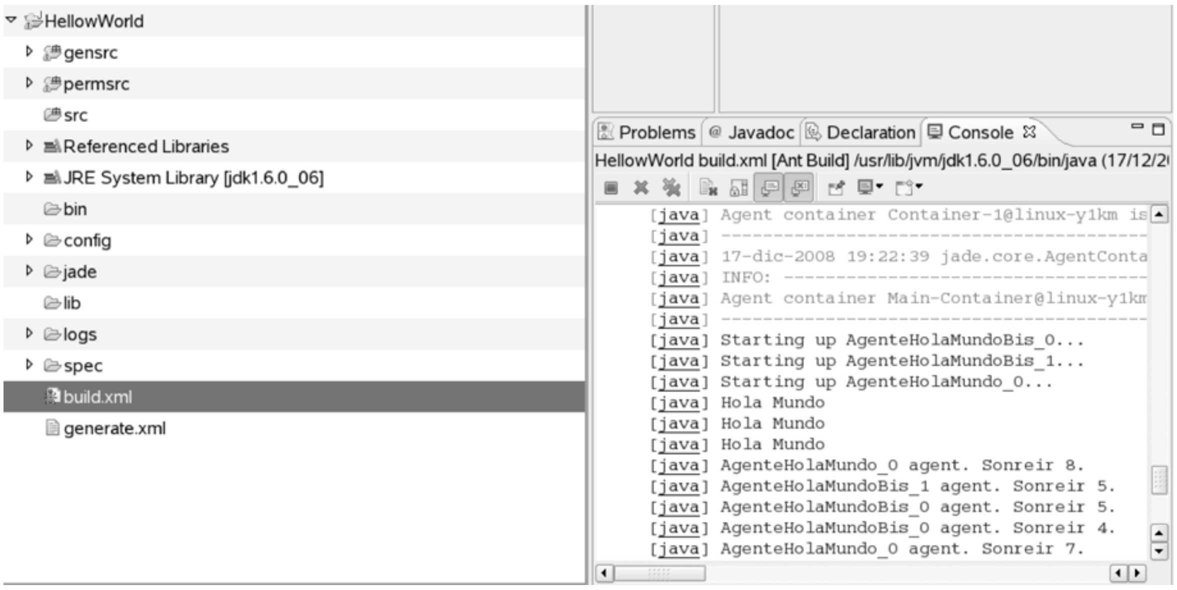
☐ Sort targets

☐ Hide internal targets not selected for execution

Target execution order:

runProdStandAlone

Figure 20: Select the target of Execution



HellowWorld build.xml [Ant Build] /usr/lib/jvm/jdk1.6.0_06/bin/java (17/12/2

```
[java] Agent container Container-1@linux-y1km is
[java] -----------------------------------------
[java] 17-dic-2008 19:22:39 jade.core.AgentConta
[java] INFO: -----------------------------------
[java] Agent container Main-Container@linux-y1km
[java] -----------------------------------------
[java] Starting up AgenteHolaMundoBis_0...
[java] Starting up AgenteHolaMundoBis_1...
[java] Starting up AgenteHolaMundo_0...
[java] Hola Mundo
[java] Hola Mundo
[java] Hola Mundo
[java] AgenteHolaMundo_0 agent. Sonreir 8.
[java] AgenteHolaMundoBis_1 agent. Sonreir 5.
[java] AgenteHolaMundoBis_0 agent. Sonreir 5.
[java] AgenteHolaMundoBis_0 agent. Sonreir 4.
[java] AgenteHolaMundo_0 agent. Sonreir 7.
```

Figure 21: Hello World Message (in Spanish, Hola Mundo)

This launches a JADE container and all deployed agents. There is no need to launch JADE in a separated console. There is no additional GUI, but it is useful for production environments (see an example of execution in Figure 21).

2.6. Hello World with Several Agents

This example is an extension of the “Hello World” example, to show how to configure a deployment of a MAS in INGENIAS.

A “Deployment” diagram defines the units (groups of agents of certain types) that conform the MAS. This example considers a MAS of three “Hello World” agents that say “Hello” to the outside world. In order to specify this with IDK, first create a Deployment diagram. In this diagram, create a DeploymentPackage with DeploymentUnitByType. In the second one, determine the type of agent and the desired number of agents (i.e. number of instances), as Figure 22 shows.

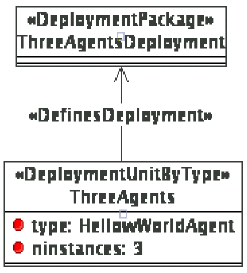
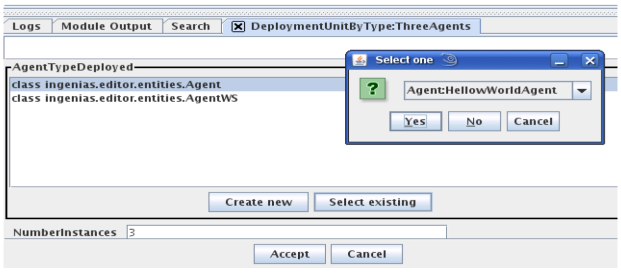


Figure 22: Deployment of three agents

To determine the type of agent to deploy, double click on the DeploymentUnitByType entity (see Figure 23), where the type of agent can be chosen from a list.



AgentTypeDeployed
class ingenias.editor.entities.Agent
class ingenias.editor.entities.AgentWS

Agent:HellowWorldAgent

NumberInstances 3

Figure 23: Selection of the content of the DeploymentUnitByType

Invoke again the code generation. The number of targets in the “build.xml” file grows.

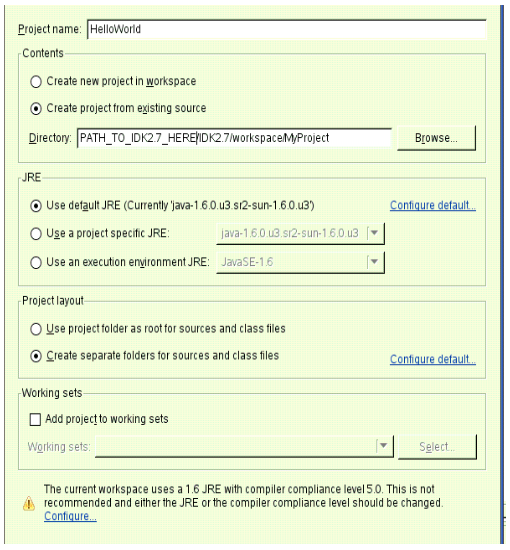
There are options to run the new deployment (see the execution results in Figure 21):

```
> ant runThreeAgentsDeployment
```

Like before, there is the option:

```
> ant runThreeAgentsDeploymentProdStandAlone
```

If using the IDK Eclipse plugin, invoke the project wizard with menu “File -> New -> Project” (see Figure 24), choose a Java project. In the next screen, choose the create project from existing source option and point to the project folder.



Project name: HelloWorld

Contents

- Create new project in workspace
- [✓ Create project from existing source]

Directory: PATH_TO_IDK2.7_HERE/IDK2.7/workspace/MyProject — Browse...

JRE

- [✓ Use default JRE (Currently 'java-1.6.0.u3.sr2-sun-1.6.0.u3')] — Configure default...
- Use a project specific JRE: java-1.6.0.u3.sr2-sun-1.6.0.u3
- Use an execution environment JRE: JavaSE-1.6

Project layout

- Use project folder as root for sources and class files
- [✓ Create separate folders for sources and class files] — Configure default...

Working sets

- Add project to working sets

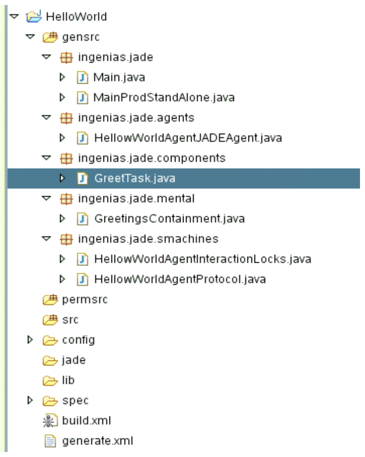
Working sets: — Select...

The current workspace uses a 1.6 JRE with compiler compliance level 5.0. This is not recommended and either the JRE or the compiler compliance level should be changed.
Configure...

Figure 24: Integration with the Eclipse Plugin

Now the different folders are accessible from Eclipse (see Figure 25):

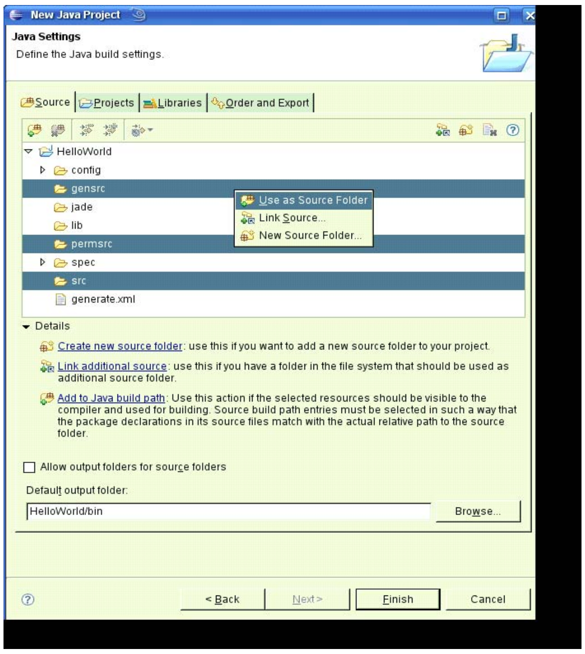
- **Gensrc** folder must not be modified, except what refers to tasks
- **Permsrc** can be modified. Modifications will not be overwritten
- **Src** folder is left to the user



```
HelloWorld
  gensrc
    ingenias.jade
      Main.java
      MainProdStandAlone.java
    ingenias.jade.agents
      HellowWorldAgentJADEAgent.java
    ingenias.jade.components
      GreetTask.java
    ingenias.jade.mental
      GreetingsContainment.java
    ingenias.jade.smachines
      HellowWorldAgentInteractionLocks.java
      HellowWorldAgentProtocol.java
  permsrc
  src
  config
  jade
  lib
  spec
  build.xml
  generate.xml
```

Figure 25: Structure in packages of the workspace of IDK

Folders to be set up as source folders are “gensrc”, “permsrc”, and “src” (right click on any folder to access the add to build path option), as one can observe in Figure 26. Define a bin folder for the binaries.


New Java Project

Java Settings

Define the Java build settings.

Source | Projects | Libraries | Order and Export

```
HelloWorld
  config
  gensrc
  jade
  lib
  permsrc
  spec
  src
  generate.xml
```

Details

Create new source folder: use this if you want to add a new source folder to your project.

Link additional source: use this if you have a folder in the file system that should be used as additional source folder.

Add to Java build path: Use this action if the selected resources should be visible to the compiler and used for building. Source build path entries must be selected in such a way that the package declarations in its source files match with the actual relative path to the source folder.

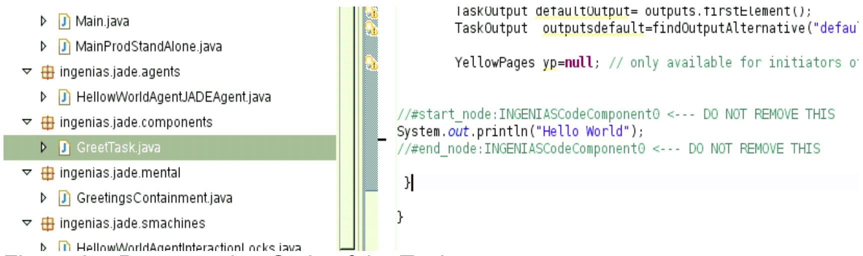
☐ Allow output folders for source folders

Default output folder: HelloWorld/bin

Figure 26: Define the source folders in Eclipse

Tasks are generated inside "gensrc/ingenias/jade/components" (see Figure 27). Look for the name of the task as it appears in the specification. Modifications have to be copied to the editor.

In the example, the task is modified to incorporate a new hello world statement. The new statement (see Figure 28) has to be copied in the specification to keep this change when the code is regenerated.


```
Main.java
MainProdStandAlone.java
ingenias.jade.agents
  HellowWorldAgentJADEAgent.java
ingenias.jade.components
  GreetTask.java
ingenias.jade.mental
  GreetingsContainment.java
ingenias.jade.smachines
  HellowWorldAgentInteractionLocks.java
```

```
        TaskOutput defaultOutput= outputs.firstElement();
        TaskOutput  outputsdefault=findOutputAlternative("defau

        YellowPages yp=null; // only available for initiators o


//#start_node:INGENIASCodeComponent0 <--- DO NOT REMOVE THIS
System.out.println("Hello World");
//#end_node:INGENIASCodeComponent0 <--- DO NOT REMOVE THIS

 }

}
```

Figure 27: Programming Code of the Task

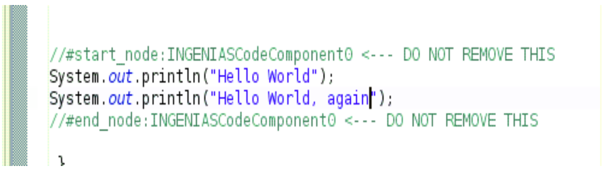
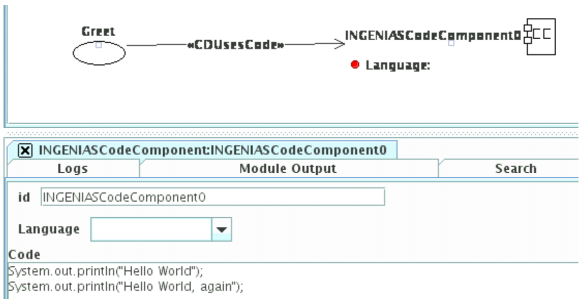

```
//#start_node:INGENIASCodeComponent0 <--- DO NOT REMOVE THIS
System.out.println("Hello World");
System.out.println("Hello World, again");
//#end_node:INGENIASCodeComponent0 <--- DO NOT REMOVE THIS

 }
```

Figure 28: Addition of a new line in the task code



INGENIASCodeComponent:INGENIASCodeComponent0

id: INGENIASCodeComponent0

Language:

Code

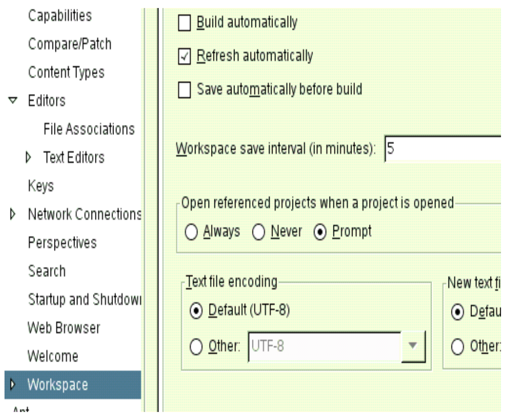
```
System.out.println("Hello World");
System.out.println("Hello World, again");
```

Figure 29: New line uploaded in the IDK

The code can be automatically uploaded to the model, with the “code uploader”. In the IDK, go to menu “Modules->Tools” and press “CodeUploader”. The results of the code upload are shown in Figure 29.

Changes different from adding code to the assigned location in the task could be removed in the next code generation. For instance, a developer cannot add a new method to the task. Instead, create these new methods in external classes and access them from the task either through public static implementations or through a singleton pattern. Be aware of including new imports in the task Java class. These imports will be removed in the next code generation. Use fully qualified names.

To avoid the use of imports and refresh automatically the project, the eclipse preferences have to be set as “Refresh automatically”, and “Use static imports”. Preferences can be selected with the menu “Window->Preferences” (see Figure 30 and Figure 31).



☐ Build automatically
☑ Refresh automatically
☐ Save automatically before build

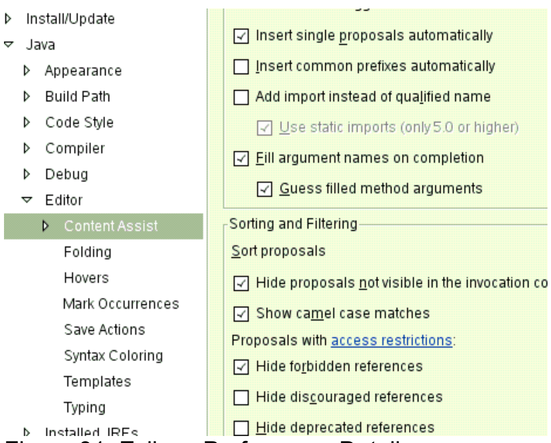
Workspace save interval (in minutes): 5

Open referenced projects when a project is opened
Always Never [✓ Prompt]

Text file encoding
[✓ Default (UTF-8)]
Other: UTF-8

New text fi
[✓ Defau]
Other:

Figure 30: Eclipse preferences



☑ Insert single proposals automatically
☐ Insert common prefixes automatically
☐ Add import instead of qualified name
☑ Use static imports (only 5.0 or higher)
☑ Fill argument names on completion
☑ Guess filled method arguments

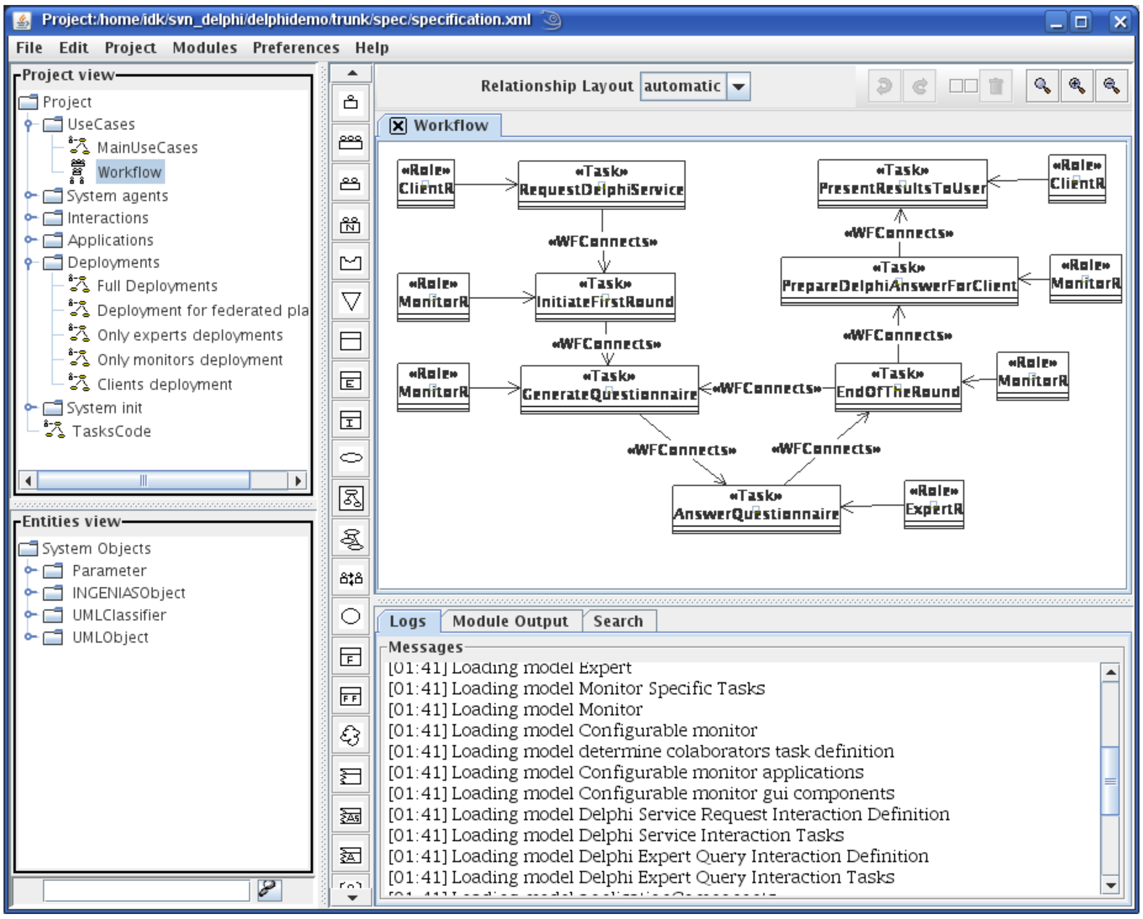
Sorting and Filtering
Sort proposals
☑ Hide proposals not visible in the invocation co
☑ Show camel case matches
Proposals with access restrictions:
☑ Hide forbidden references
☐ Hide discouraged references
☐ Hide deprecated references

Figure 31: Eclipse Preferences Details

3. The graphical editor

The main purpose of the graphical editor (Figure 32) is to create and modify the specifications of multi-agent systems (MAS). A MAS specification is a set of diagrams that represent the different views of the 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 *modules* (which are described in section 5).



```
[01:41] Loading model Expert
[01:41] Loading model Monitor Specific Tasks
[01:41] Loading model Monitor
[01:41] Loading model Configurable monitor
[01:41] Loading model determine colaborators task definition
[01:41] Loading model Configurable monitor applications
[01:41] Loading model Configurable monitor gui components
[01:41] Loading model Delphi Service Request Interaction Definition
[01:41] Loading model Delphi Service Interaction Tasks
[01:41] Loading model Delphi Expert Query Interaction Definition
[01:41] Loading model Delphi Expert Query Interaction Tasks
```

Figure 32. A screen shot of the editor

3.1. Parts of the editor and related operations

Figure 33 shows the different parts of the editor. Each part is described in the following sections.

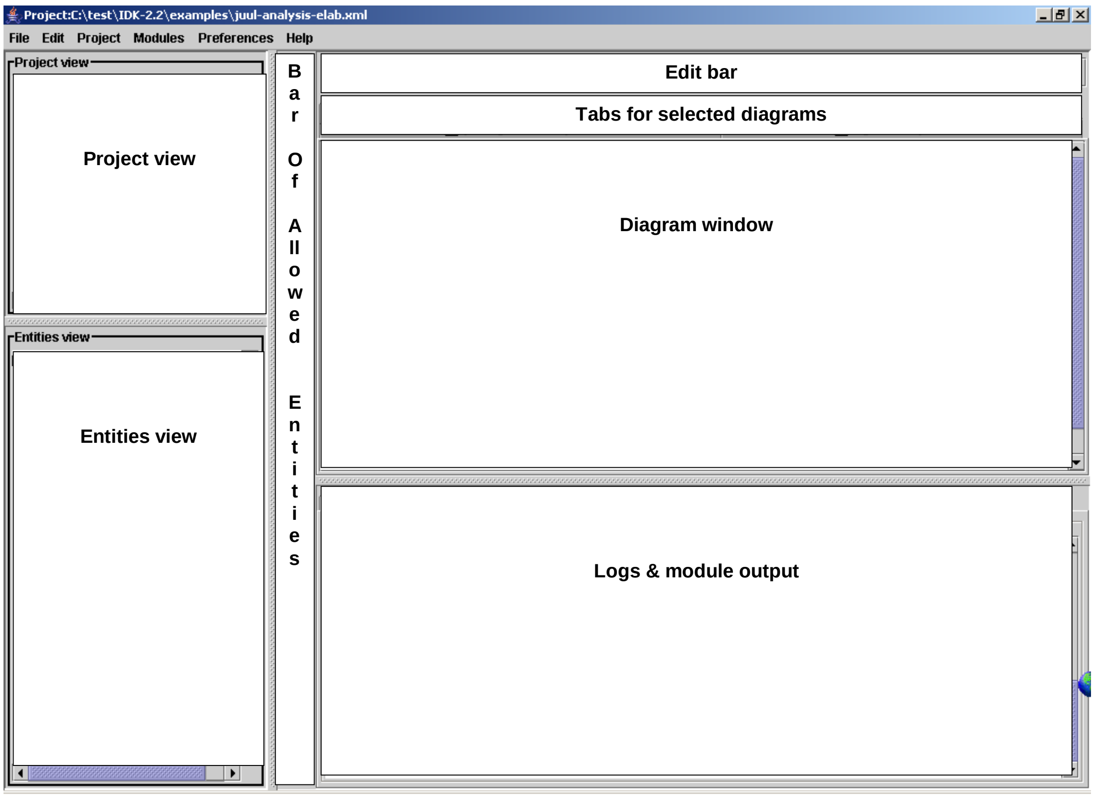


Figure 33. Parts of the editor

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**. One can drag and drop a diagram into a package or a package into another package.
- **Change the order of siblings in the project**. Select a node and use keys **Q** and **Z** to modify the order.
- **Open a diagram**. By double left clicking on the diagram.
- **Create a package**. To create a package, first select a package, then right click with 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 right click with 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 right click with the mouse and select *remove package/diagram.* This will not delete the entities contained in the diagram, but it will remove all defined relationships. Objects created for the relationships will not be deleted either.
- **Rename**. Select the diagram, and then right click with 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.

- **Copy and paste.** Use the Ctrl+C to copy and Ctrl+V to paste, for both entities and relationships. The buttons in the tool bar can also perform the same operation. Paste operation will preserve the location of the object. So, it may be the case that, on pasting an entity, one doesn't see it in the actual screen. In this case, one should try to scroll to the same position this entity had in the previous diagram; one can also zoom out to better locate it.
- **Select all elements.** Use Ctrl+A to select all elements in the diagram. This is useful to reallocate the elements in the diagram.

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.
- **Search.** It shows in the Logs & module output a list of diagrams containing the property. One can press several times in the same link to visit each occurrence of the entity in the same diagram. See Figure 4 for an example of the result of a search.

The entities view incorporates a search field that can be used to locate entities in the tree. This feature is shown in Figure 34. One can type the name of the entity and hit enter at any moment. The first occurrence of the entity in the tree will be found. Hitting enter again will find a second occurrence. Once found, a possibility is to right click in the entity and choose **search**. This will show which diagrams this entity appears, as Figure 35 shows. Clicking several times on each link (right side of the picture) will find occurrences of the entity in the same diagram. The entity will be selected by default so one can better locate the target entity among the set of existing entities.

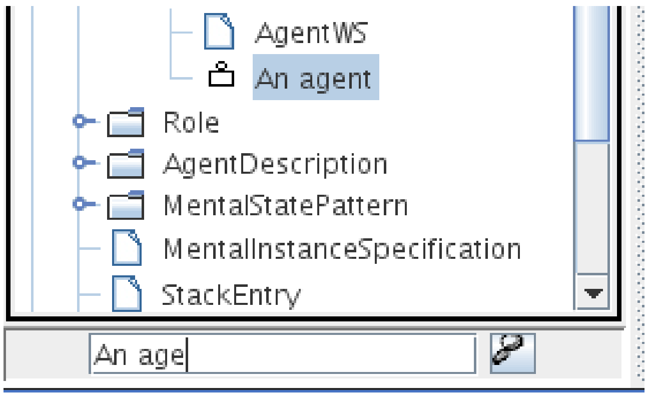


Figure 34. Searching an entity in the entities view.

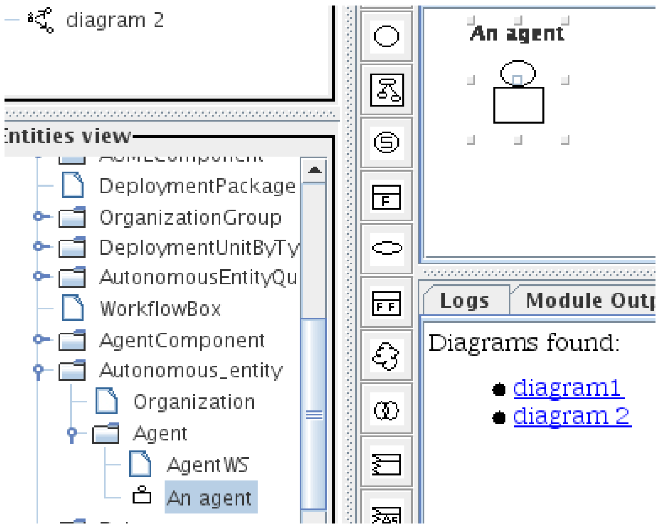


Figure 35. Looking for occurrences of an entity in different diagrams

3.1.3 Diagram editor

The Diagram editor of IDK consists of three parts:

- The *Editor Bar*, with common operations for the edition (undo, redo, zooming, copy/paste, etc.). Undo and redo operations are limited to modifications of the location of the entity. Paste and remove operations cannot be undo.
- The *Bar of allowed entities*, which is specific for each type of diagram. It has buttons that allow creating instances of particular entity types in the current diagram. Normally, all entity types that are allowed for a particular diagram appear in the bar. If the number of possible entities exceeds the visible height of the bar, arrow buttons will appear at the top and at the bottom. These buttons will permit to scroll up and down the list of buttons.

  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:

- **Show the diagram in the project view.** In any diagram, the user can right click in the background and request to show the location of the diagram in the project view (see Figure 36). The diagram will be selected by default, perhaps scrolling to it if necessary.

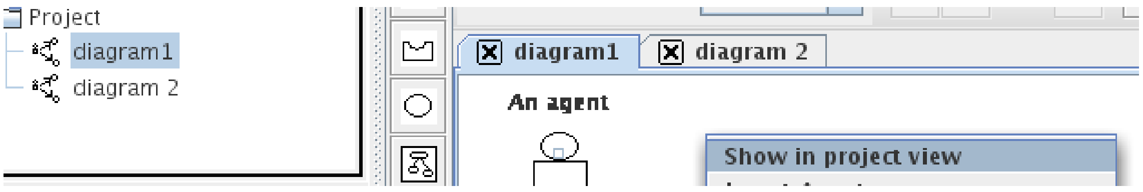
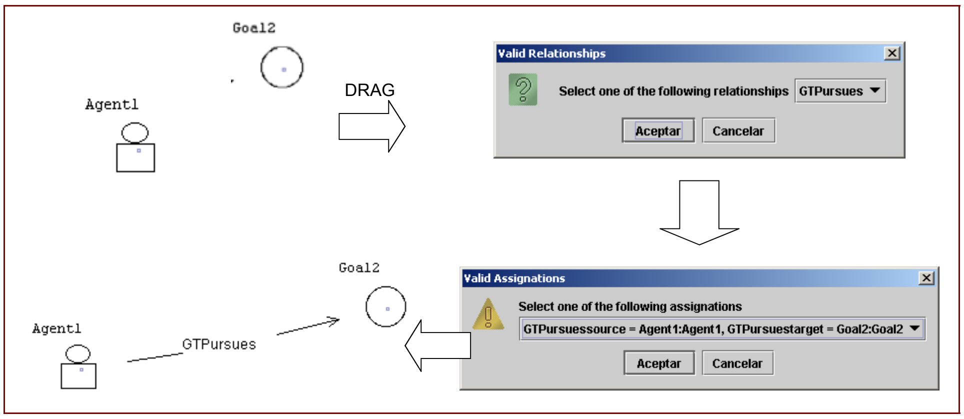


Figure 36. Locating the current diagram in the project view

- **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 37 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. If more than one connection possibility exists, 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). In case of several possibilities, the user has to select one. 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. This last step will be omitted if there is only one possibility.



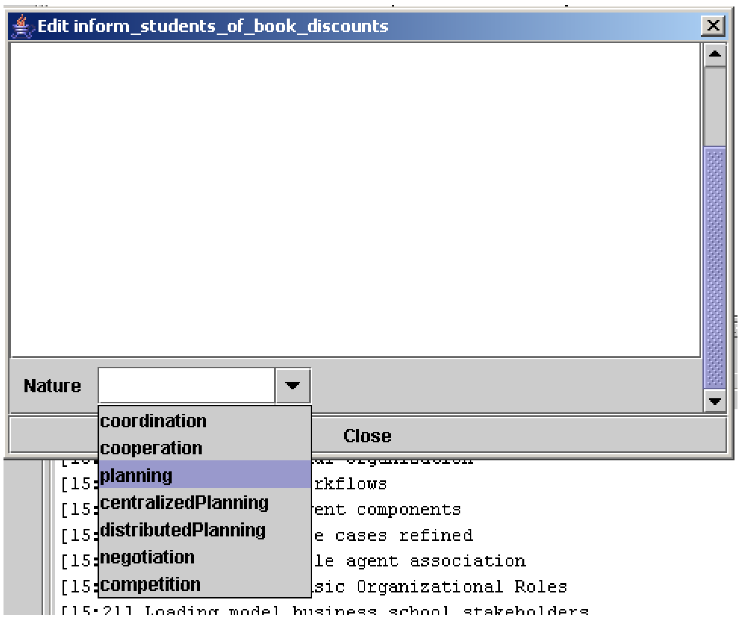
Select one of the following relationships GTPursues

Select one of the following assignations
GTPursuessource = Agent1:Agent1, GTPursuestarget = Goal2:Goal2

Figure 37. Steps to follow to create a relationships among two entities

A second possibility is to select several entities (e.g., with shift and mouse right button), and selecting the connect option of the contextual menu (this menu appears when clicking the mouse right button in an area outside of the selected entities).

- **Adding a new entity to an existing relationship**. Some relationships accept more than two entities, i.e., they are n-ary. When a relationship is already created and the user wants to add another entity, the process is simple. First, the user moves the mouse to the relationship to which the user wants to connect the entity until the mouse icon changes. Then, the user drags the mouse towards the entity the user wants to connect, and the user releases the mouse. Following, a window will appear showing what kind of role will play the new entity in the relationship. The user selects one and accept the new type. If the entity cannot be accepted by the relationship, due its type or the cardinality of the relationship, an error dialogue will appear



```
[15:         rkflows
[15:         ent components
[15:         e cases refined
[15:         le agent association
[15:         sic Organizational Roles
[15:21] Loading model business school stakeholders
```

Figure 38. An 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.** One can write whatever is necessary. 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 38). A value has to be selected.

- **Diagram reference fields.** This field allows to refer to other diagrams (see Figure 39). 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, one can press *show selected.*

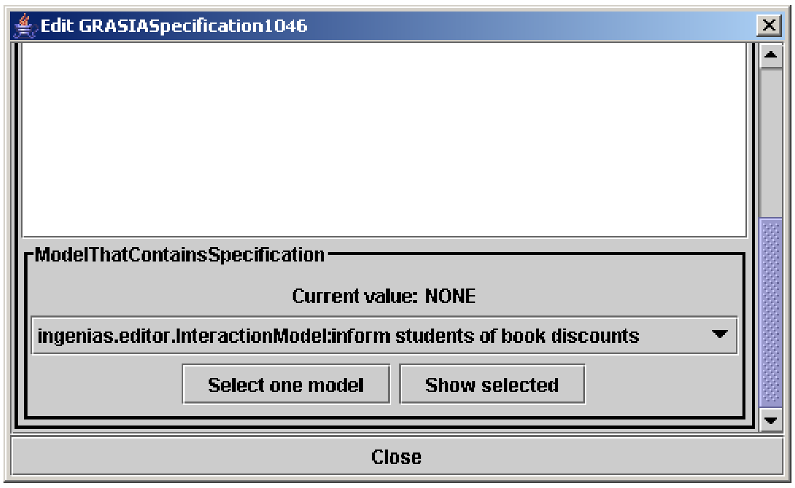

Edit GRASIASpecification1046
ModelThatContainsSpecification
Current value: NONE
ingenias.editor.InteractionModel:inform students of book discounts
Select one model
Show selected
Close

Figure 39. 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 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 dialogue window will appear with a combo box showing valid already defined entities that could be used. Select one and press *yes.*

    - **Add new.** A dialogue 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 40) will show possible types that could be allowed in that part of the definition. One of the types must be selected and, afterwards, one of the available buttons must be pressed. With the *Create new* button, a new instance of the selected type will be created and associated directly with this field. The new instance will also appear in the *Entities View.* With the *Select Existing* button, this field will be associated with an existing entity. Existing entities of selected type will be shown in a dialogue window in form of a combo box.

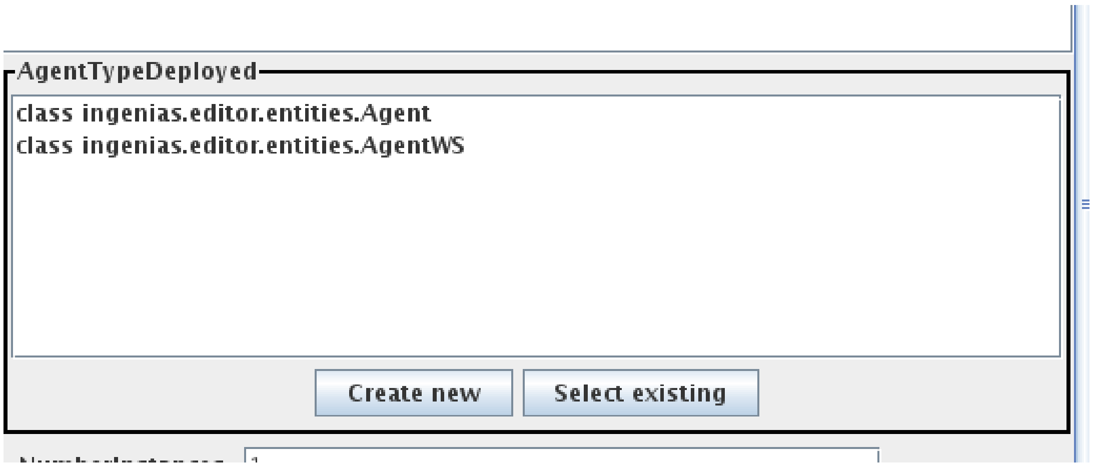

AgentTypeDeployed
class ingenias.editor.entities.Agent
class ingenias.editor.entities.AgentWS
Create new
Select existing

Figure 40. An editing window for an entity showing how to modify single value entity field.

Once selected or created a new entity, the original dialogue changes to the one shown in Figure 10. From this situation, it is possible to undo the previous selection. This is achieved with the **Unlink** button. If this button is pressed and the screen is resized, the dialogue will revert to the one shown in Figure 41.

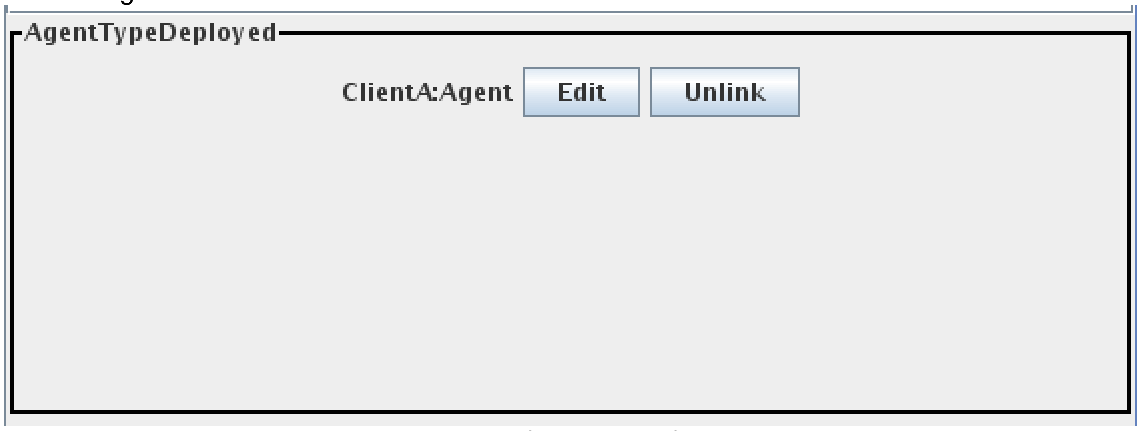


Figure 41. An editing window for an entity after having selected one.

- **Changing the icon of an entity.** Some entities have different associated views, which can be selected by right clicking on an element and going to the *views* option. There, available views will be shown (see Figure 42).

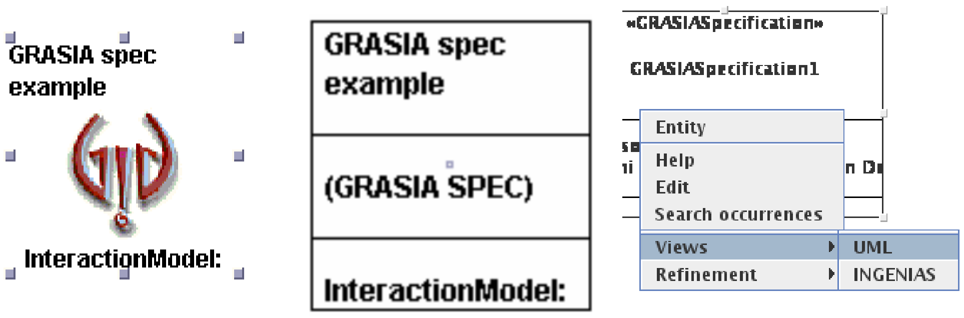


Figure 42. Different views associated with a GRASIA Specification. First view is the *icon* view, and the second the *box* view

- *Edit bar.* This bar offers options to
  - **Zoom in / zoom out**. 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 *the user is responsible for allocating the entity*. By default, the Automatic layout is used.

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.

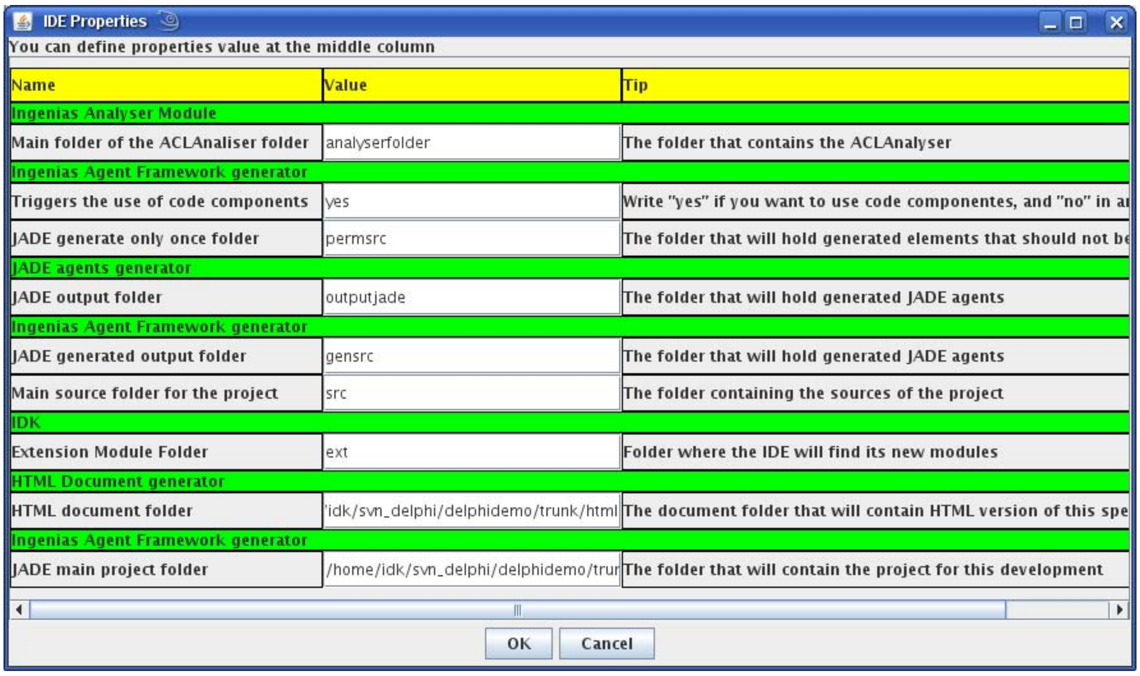
3.1.5 Main menu

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. The IDK also provides 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. Besides, these functions, there is this one which is important:
  - **Import.** It merges another specification into the current one. It is useful to reuse previous work.
- **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 dialogue 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, png, svg, and eps*. We advise to use eps, since it ensures the diagram quality and it permits to use these files with Latex. The svg format is not compatible for all browsers.

- ***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 dialogue window that will be shown.

  Also, this menu provides access to the *project properties window*. This window, see Figure 43, 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. It is worth noting that one can only modify the second column, the one titled *values*. It is made of text fields, so it is easy to paste or modify the text. The **ok** button must be pressed when finished.



IDE Properties

You can define properties value at the middle column

Name	Value	Tip
Ingenias Analyser Module		
Main folder of the ACLAnaliser folder	analyserfolder	The folder that contains the ACLAnalyser
Ingenias Agent Framework generator		
Triggers the use of code components	yes	Write "yes" if you want to use code componentes, and "no" in a
JADE generate only once folder	permsrc	The folder that will hold generated elements that should not be
JADE agents generator		
JADE output folder	outputjade	The folder that will hold generated JADE agents
Ingenias Agent Framework generator		
JADE generated output folder	gensrc	The folder that will hold generated JADE agents
Main source folder for the project	src	The folder containing the sources of the project
IDK		
Extension Module Folder	ext	Folder where the IDE will find its new modules
HTML Document generator		
HTML document folder	'idk/svn_delphi/delphidemo/trunk/html	The document folder that will contain HTML version of this spe
Ingenias Agent Framework generator		
JADE main project folder	/home/idk/svn_delphi/delphidemo/trur	The folder that will contain the project for this development

OK Cancel

Figure 43. Properties of a project. The user can only edit 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 44). The *tools* entry contains modules whose main purpose is not generating code but analysing 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.

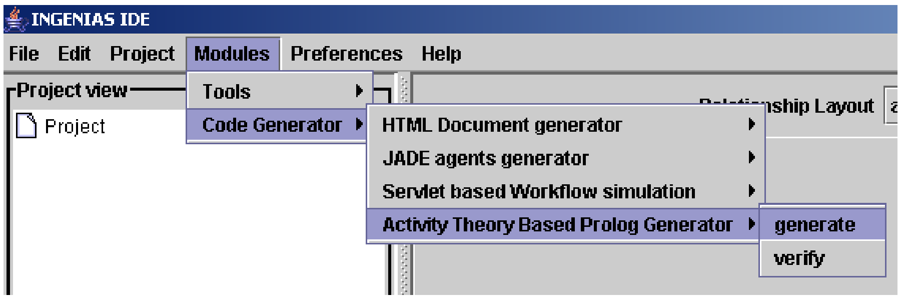


Figure 44. Module list available in the 2.2 version of the IDK

- **Preferences menu.** This menu modifies the way the editor works. There are many possibilities.

  1. Resize all entities within the current diagram. It checks the size of all entities in the current diagram and resizes them so that all information is visible.
  2. Resize all entities in all defined diagrams. It does the same as the previous option but on all diagrams. This option is useful when switching views.

3. Eliminate overlap. It checks the current location of each individual entity. If there is an overlap with any other entity, see Figure 45, a repulsion vector is computed and applied, so that both entities move in opposite directions. As a result, the overlap is closer to elimination. The operation should be invoked several times, since the entities are moved just a little each time. So, instead of going to the menu, the user may prefer pressing directly the key **F3** and observe the result.
4. Edit Properties mode. It refers to where the edit properties dialogue appear. It can be either as embedded dialogue in the bottom part of the editor or as a pop-up.
5. Modelling language. It permits to modify the way the entities appear in the diagram. There are two possible views: INGENIAS and UML. The INGENIAS view uses icons to represent each entity, according to the INGENIAS notation. The UML view uses the known representation in form of Boxes (see Figure 46).

   Enable INGENIAS view from now on. All entities created from now on will be shown initially with icons coming from INGENIAS notation.

   Enable UML view from now on. All entities created from now on will be shown initially with UML notation

   Switch to UML view. Change all entities in all diagrams to UML notation.

   Switch to INGENIAS view. Change all entities in all diagrams to INGENIAS notation.

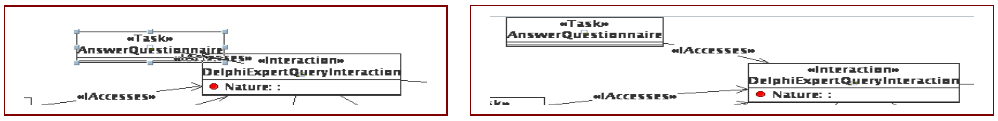


Figure 45. Overlapping elimination

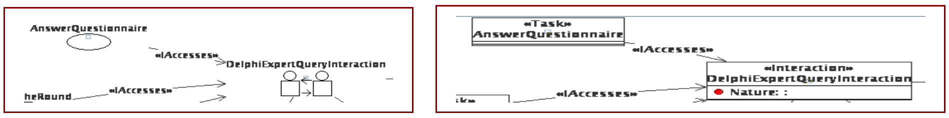


Figure 46. To the left, INGENIAS view and to the right UML view.

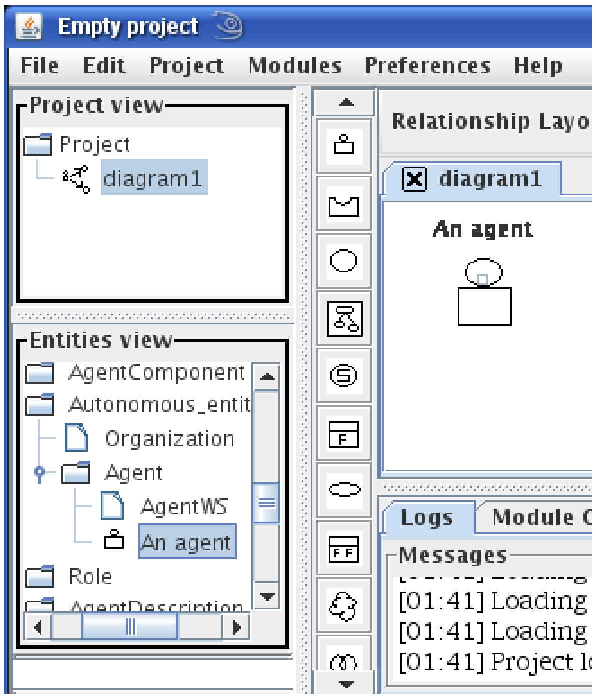
- **Help menu**. It provides access to a summarised version of this document (*tool manual* option), the credits, and the possibility of forcing a garbage collection to optimise memory usage (*Force GC* option). Please note that according to SUN specifications, it seems that calls to the garbage collector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

3.2. Hints for working with the IDK editor

In general, the use of the editor is quite straightforward, but here are some recommendations for working with it:

1. **It is not necessary to create a new object each time.** For instance, one can create an agent instance so one can observe the situation depicted in Figure 47. Pay attention to the **entities view** where the instance of the agent appears. Now, assume the user wants to reuse this entity into another diagram. There are several possibilities
   a. Copying the entity directly from the diagram. To do this, just select the entity and hit **ctrl+c**, then go to the target diagram and press **ctrl+v**. As an alternative, the user can use the copy&paste buttons from the application or the copy and paste options from the **Edit menu**.
   b. Going to the second diagram. It will be blank at the beginning. Then right click on **an agent** icon in the **entities view**. The pop-up menu presented in Figure 48 will appear. Choose **add to current diagram**.

After any of the previous steps, the situation shown in Figure 49 will take place, two diagrams with a copy of the entity, and only one instance in the **entities view**.



```
[01:41] Loading
[01:41] Loading
[01:41] Project l
```

Figure 47. Organization of work with packages

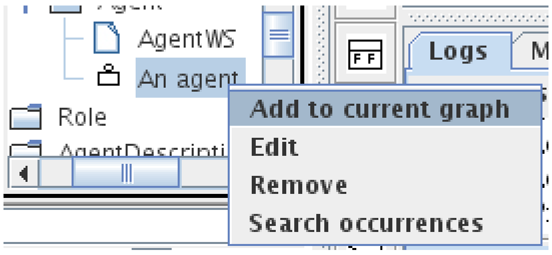


Figure 48. Adding an entity to a diagram

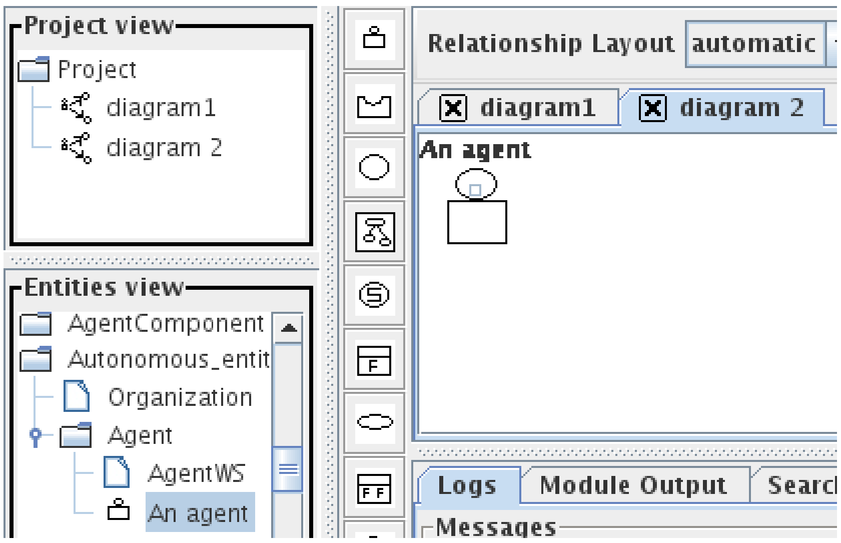


Figure 49. Adding an Entity to a diagram

2. **Think in advance which diagrams will be necessary 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, diagrams are necessary to describe how an agent performs a task A, create a folder with a label something like *specification of task A execution* (see Figure 50)

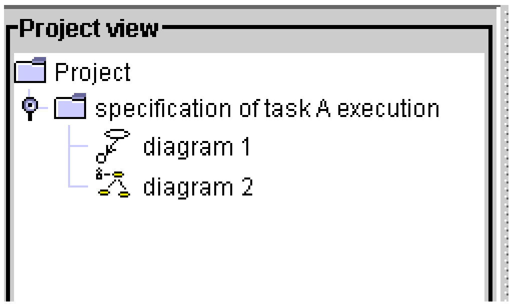


Figure 50. Organization of work with packages

3. **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.
4. **In case the user gets lost and doesn't find any entity**. The user can try using the search utility of the entity view. This will allows one to search the entity. To see what diagrams it is included into, right click on the entity and select **search**.
5. **Do not be afraid if there are too many diagrams.** This is usual in any conventional development. For this reason, this manual recommends to start using packages (see Figure 51) from the beginning. Anyway, the user can always create them later and use the drag&drop feature to rearrange them.

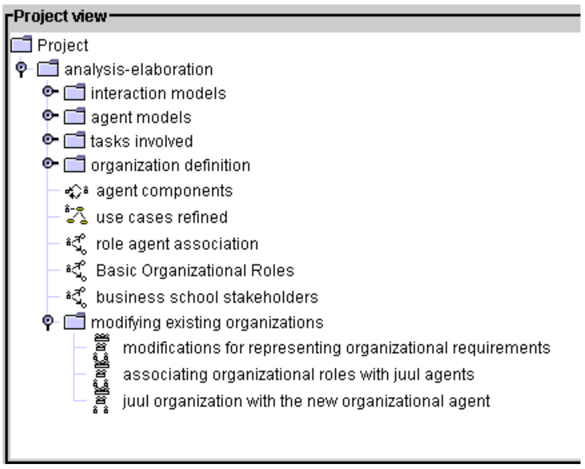

Project view
Project
analysis-elaboration
interaction models
agent models
tasks involved
organization definition
agent components
use cases refined
role agent association
Basic Organizational Roles
business school stakeholders
modifying existing organizations
modifications for representing organizational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

Figure 51. The number of diagrams can grow easily when getting into details. See how a package structure helps to manage them

6. **Copy diagrams to the clipboard.** Remember that the editor allows one to copy a diagram to the preferred text editor by copy&paste option (**Edit menu**, not the ctrl+c/v key). The quality is rather good, but perhaps one prefers to save the image to a file, so that one can include the image in a latex file or similar. In this case, the *eps* format is recommended, which is the one guaranteeing the best quality.

4. Automatic Backups

IDK makes automatic backups periodically, so, if there is some unexpected crash while using the IDK, it is still possible to recover the work done so far. Proceed by following these steps:

1. **Do not open the IDK yet**. When starting the IDK, the backup is overwritten, so one may loose the data.
2. Go to the home folder and look for a folder called *.idk.*
3. Enter that folder and inspect the files. They are labelled idkbackupX.xml, being X a number between 0 and 9. These numbers mean there are 10 security copies, each copy being made 5 minutes after the previous one. So, there is a backup of the last 50 minutes of work. After these minutes, the files are overwritten with new ones.
4. Select the one with the newest modification date and copy it to the preferred folder. The user can also copy all of them.
5. Open the IDK and load the copied file to check it is the intended one.
6. Save it with another name to resume the work.

One should never open directly a file stored in the .idk folder with the IDK. This will cause an undesirable situation when trying to save the work in a file that is being accessed by the automatic backups functionality. For this reason the user is advised to copy the file first.

5. Modules

Modules (see Figure 52) are programs that process specifications and produce some output, which can be:

- **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, if they have certain properties, if they respect some special semantics, defined by the developer, 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 modify entities in the diagrams, or insert/delete diagrams. This feature is useful to define personal assistants that interact with the tool.

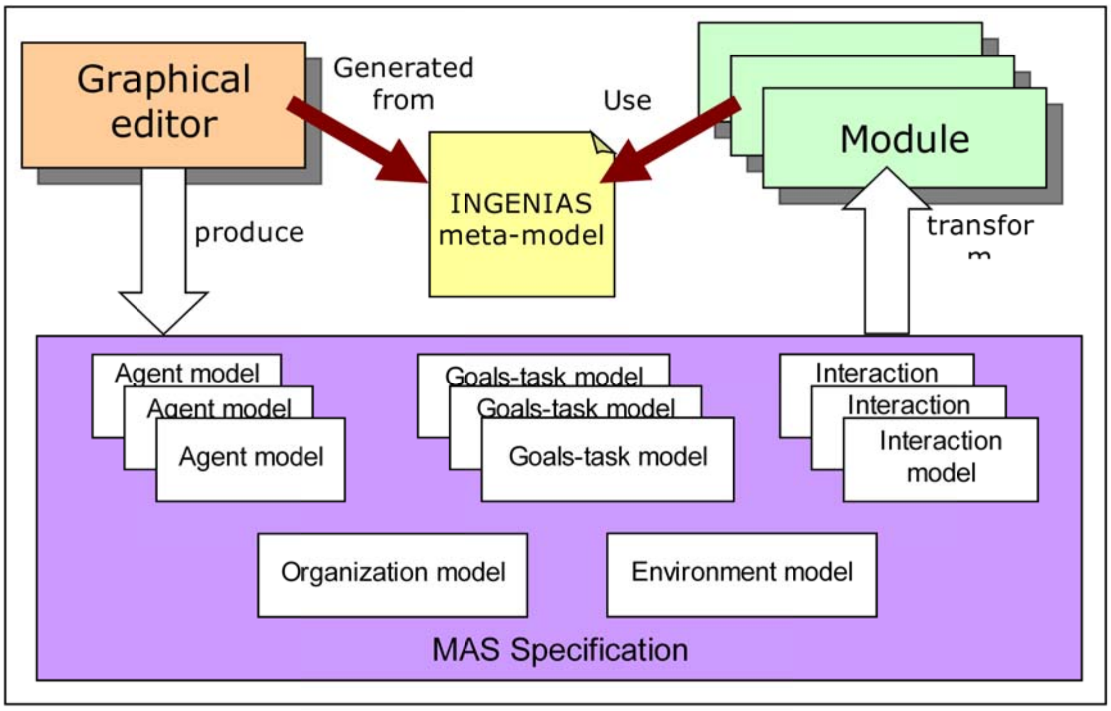


Figure 52. Relationship between the IDK, models and components

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 (see Figure 53). 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.

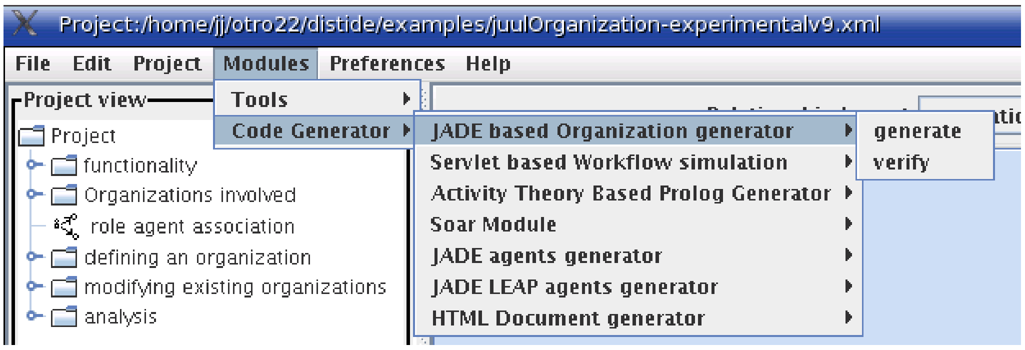

Project:/home/jj/otro22/distide/examples/juulOrganization-experimentalv9.xml
File
Edit
Project
Modules
Preferences
Help
Project view
Project
functionality
Organizations involved
role agent association
defining an organization
modifying existing organizations
analysis
Tools
Code Generator
JADE based Organization 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 53. Invocation of a module in IDK

5.2. Development of new modules

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

Developing a module implies several steps:

1. **Producing a prototype of the application.** A developer would centre into one or more features, easy to implement if possible. These features would be expressed with INGENIAS diagrams that an analyst would have produced.
    a. **Tools:** a conventional development environment (depending on the programming language and target platform, in the case of code generation).
    b. **Outcome:** a prototype that realizes a part of the INGENIAS specification. In other words, the designer has to create the programming code that is expected to be generated for the piece of the INGENIAS specification.
2. **Marking 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. **Outcome:** 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).
    a. **Tools:** a conventional J2SE development environment
    b. **Outcome:** 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 created in a specific folder where the IDK Editor can load it.
    a. **Tools:** J2SE and *ant* (http://ant.apache.org). The J2SDK jar tool generates the jar, and the *ant* tool executes the appropriate *ant* task that perform the compilation and copy of the source and binary files.
    b. **Outcome:** 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.

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

    b. **Outcome:** 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:

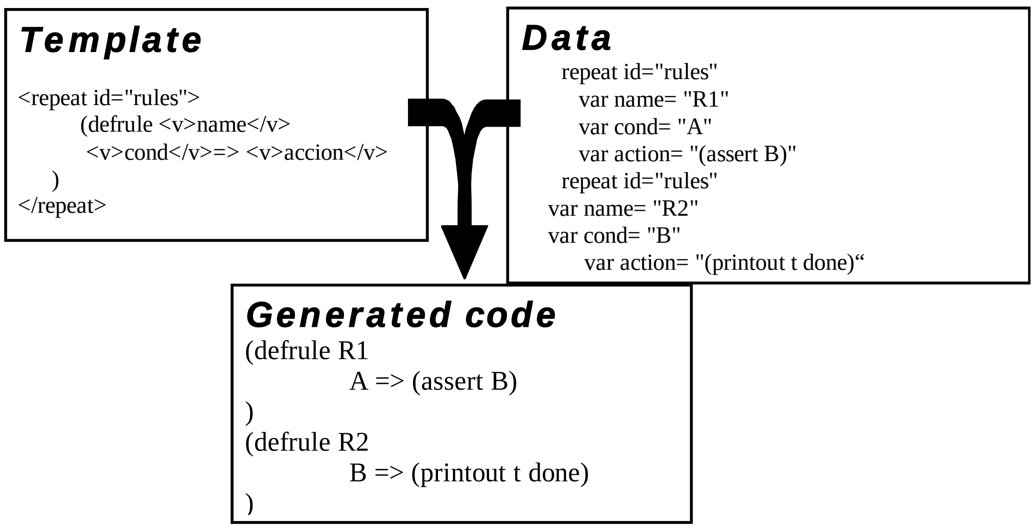
    a. *Step 2.* If there is new code that was not marked up before.

    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 54 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.



Template

```
<repeat id="rules">
        (defrule <v>name</v>
         <v>cond</v>=> <v>accion</v>
    )
</repeat>
```

Data

```
    repeat id="rules"
       var name= "R1"
       var cond= "A"
       var action= "(assert B)"
    repeat id="rules"
  var name= "R2"
  var cond= "B"
       var action= "(printout t done)“
```

Generated code

```
(defrule R1
            A => (assert B)
)
(defrule R2
            B => (printout t done)
)
```

Figure 54. 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, the *ingenias.editor.extension. BasicCodeGeneratorImp* class must be extended. For modules of the second type (e.g., a verification tool), the *ingenias.editor.extension.BasicToolImp* class must be extended.

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

MAS specifications are structured as graphs with the elements and relationships shown in Figure 55. This data structure, which is an interpretation of the GOPRR model, needs to be known when building an IDK module in order to traverse specifications. The types of elements that form a MAS specification with the IDK are:

- *Project*, represents a MAS specification (in IDK each MAS is developed in a project). Each MAS project consists of several diagrams.
- *Graph*, represents a diagram. A graph has several entities, which are linked by relationships. A graph may have also several attributes.
- *Entity,* represents an element of a specification. Each entity has several attributes. Some entities may reference to diagrams.
- *Relationship,* to connect entities. Each relationship may have two or more ends (*GraphRole*). Each role may have also attributes.
- *Attribute,* can be entities or make reference to other graphs.

Note that 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.

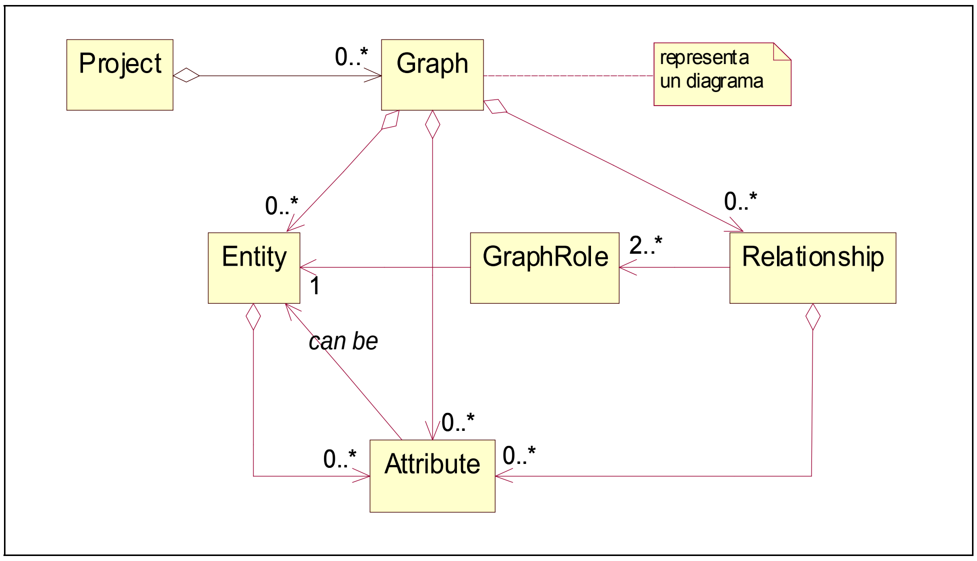


Figure 55. Logical view of the data stored in the IDK

The access to this structure is controlled by a set of interfaces, which are shown in Figure 56. These interfaces take advantage of the commonalities of the elements of Figure 55, 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 the module is expected to be executed outside the editor directly over an specification file generated by the IDK Editor, the programmer should include the following code into the main method:

```
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: we allow repetitions. 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 it connects*" or as complex as a traversal with the purpose of generating JADE code.

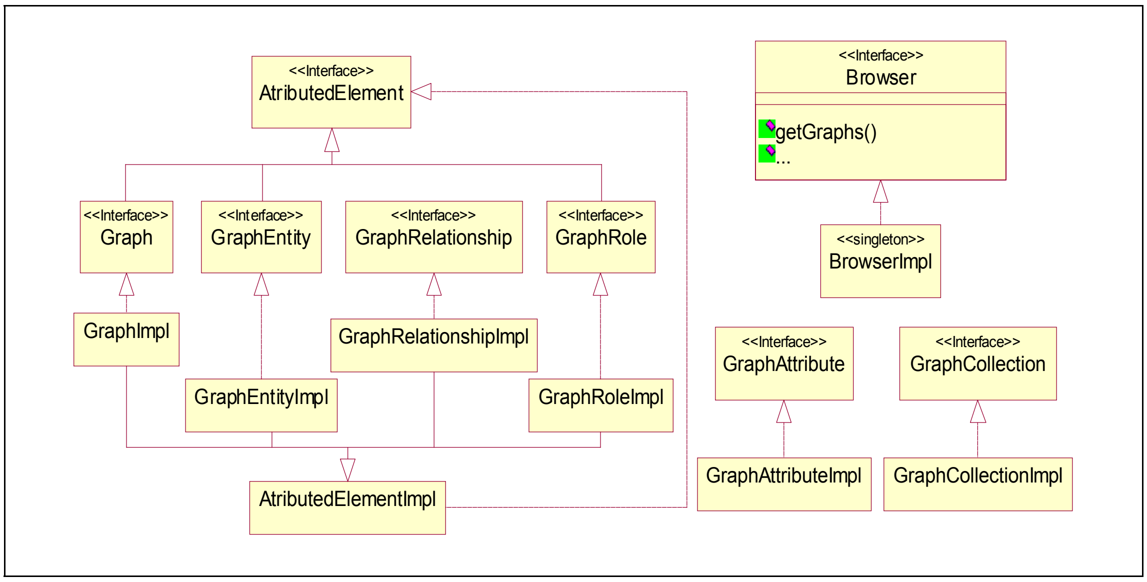


Figure 56. Interfaces provided to access the data stored as XML

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 57. 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 57. DTD for extracted information

Tags from Figure 57 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
    - **file**. It is the name of the file. It can contain other tags as well.
    - **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 the < 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 57 and data to fill the template *v* and *repeat* tags. The data that feeds the code generator has the structure shown in Figure 58. 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.

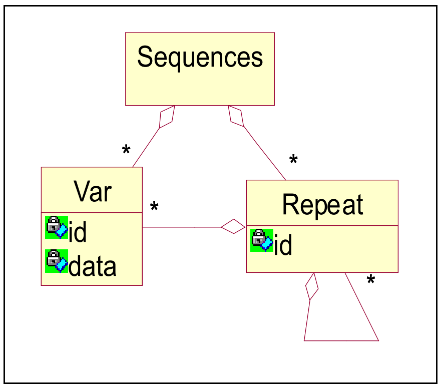


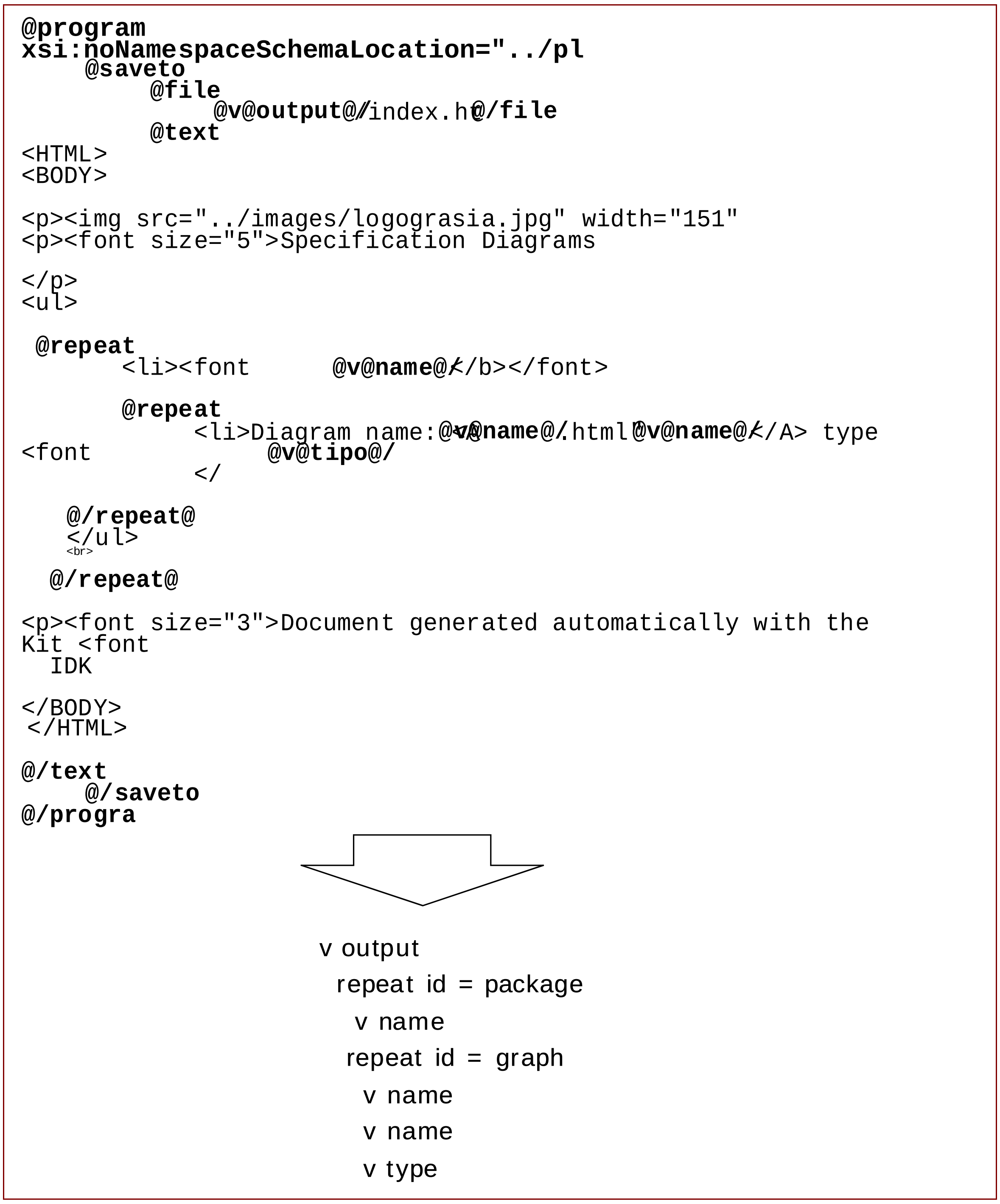
Figure 58. UML Description of the data structure

As Figure 58 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 59 shows the data structure needed to instantiate a template of the html code generator module included in the IDK distribution. The template corresponds to an *index.htm* file.



```
@program
xsi:noNamespaceSchemaLocation="../pl
    @saveto
        @file
            @v@output@/index.ht@/file
        @text
<HTML>
<BODY>

<p><img src="../images/logograsia.jpg" width="151"
<p><font size="5">Specification Diagrams

</p>
<ul>

 @repeat
      <li><font     @v@name@</b></font>

      @repeat
           <li>Diagram name:@v@name@/.html"@v@name@</A> type
<font          @v@tipo@/
           </

   @/repeat@
   </ul>
   <br>

  @/repeat@

<p><font size="3">Document generated automatically with the
Kit <font
  IDK

</BODY>
 </HTML>

@/text
    @/saveto
@/progra
```

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

Figure 59. Information structure extracted from the template

A similar Java structure with the classes from Figure 58 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).

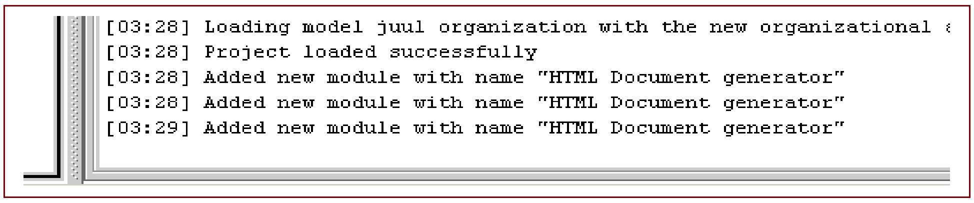
5.2.4 Deploying a module

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 60). Each time a module is deployed, 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 60. 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, listed following (bolder and italics represent comments inserted to facilitate understanding):

```
Change the location attribute to the path to the 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, one could get a personalized task to compile and deploy the 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 the 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.

6. 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 that several references point to the same object.
- **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 life cycle.** 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.

6.1. 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 parametrization 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.

6.2. Meta-modelling

Though there may be previous interpretations of what meta-modelling 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.

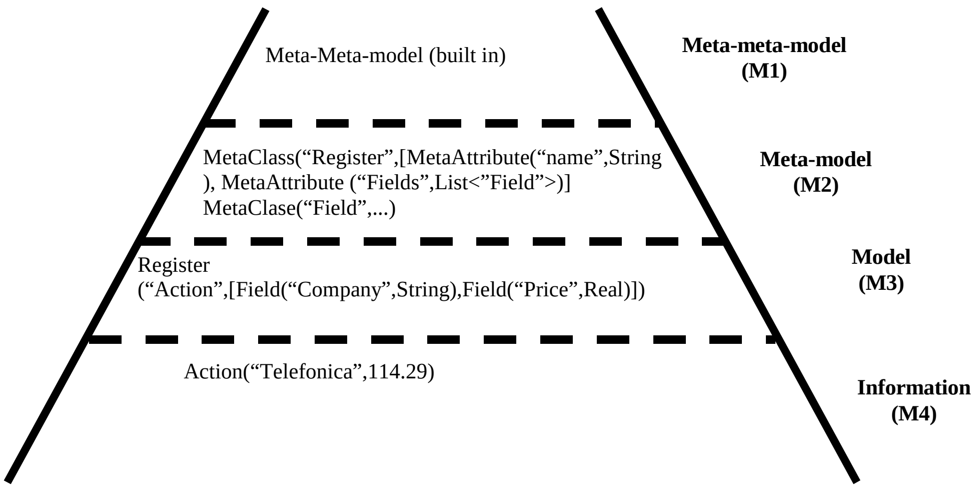
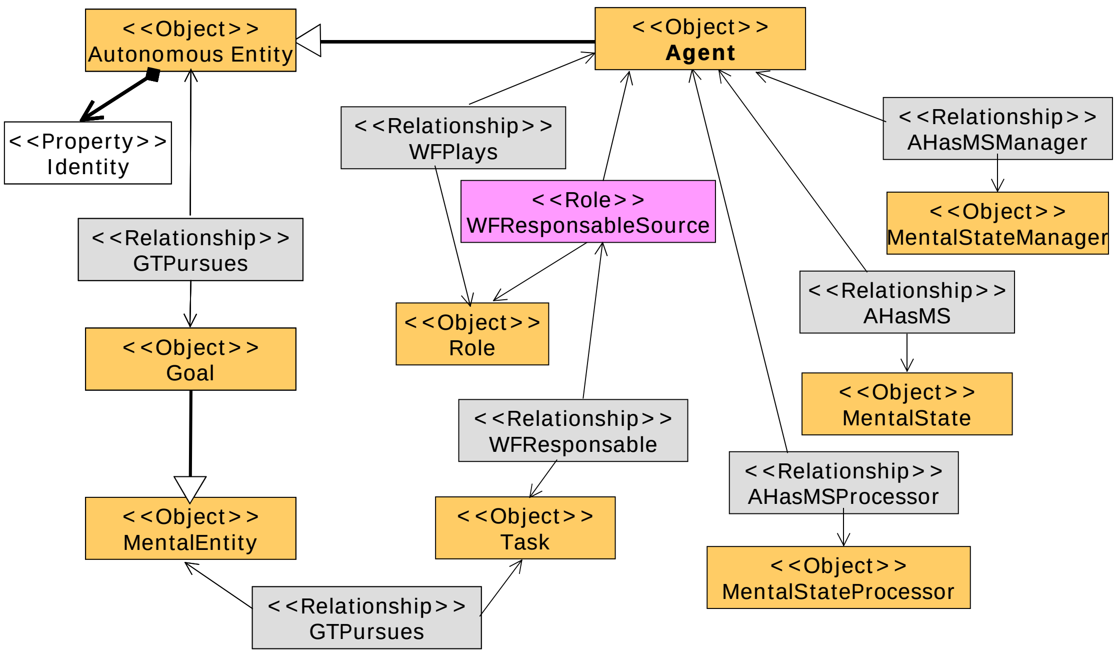

Meta-Meta-model (built in)
Meta-meta-model (M1)
MetaClass("Register",[MetaAttribute("name",String), MetaAttribute ("Fields",List<"Field">)] MetaClase("Field",...)
Meta-model (M2)
Register ("Action",[Field("Company",String),Field("Price",Real)])
Model (M3)
Action("Telefonica",114.29)
Information (M4)

Figure 61. Levels when meta-modelling according to (OMG, 2000)

In INGENIAS we use the schema of Figure 61 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 62. 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 63, 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 62 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*.

6.3. INGENIAS Meta-models

INGENIAS meta-models define five viewpoints in order to define a MAS (see Figure 63).

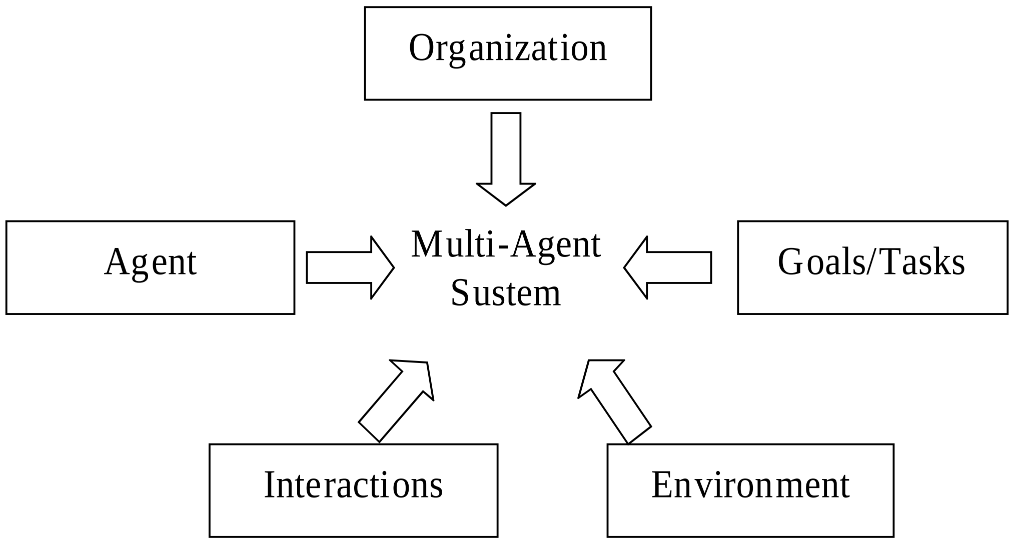


Figure 63. MAS specification viewpoints

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. The developer defines the organization attending the expected groups of agents. Functionality is determined when defining the goals of the organization and the expected workflows.
- **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 the 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.

6.4.FAQ ABOUT INGENIAS

- Why choose INGENIAS instead other agent-oriented software engineering methodologies (such as Prometheus, Tropos, PASSI, etc) ?
    - INGENIAS is supported by IDK, and it generates fully functional MASs. INGENIAS is continuously supported by the Grasia research group, and each year there is a new stable version, and from svn, one can get the latest improvements. INGENIAS include an specific debugging platform, where the engineers can see the mental state of the agents, its interactions, executed tasks an so on. INGENIAS and IDK won an award for the best academic demo in the AAMAS'08.
- 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 diagrams? It depends on the purpose. Here some hints are provided:
    - Agent diagrams.
        - These kinds of diagrams are recommended when expressing an intermediate state of the agent. Figure 64 presents an example.

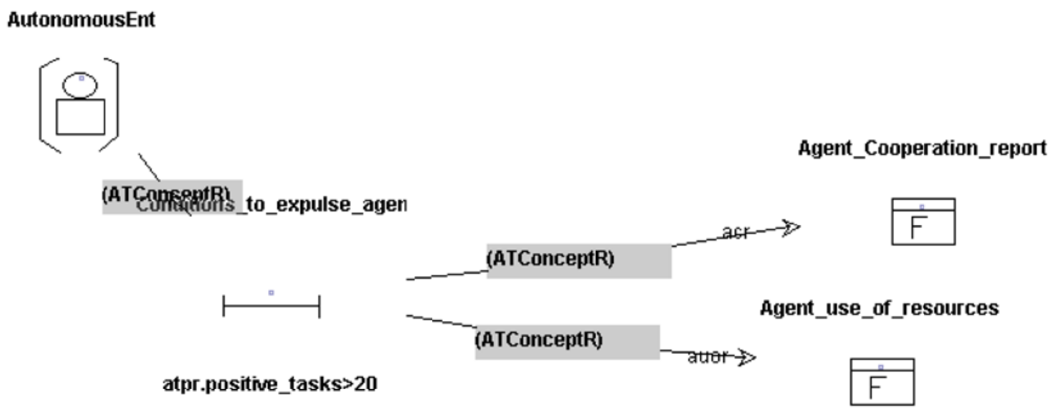


Figure 64. Mental state required to determine if an agent should be excluded or not

        - They are also recommended when the initial setting of the agents need to be expressed.
    - Interaction diagrams
        - How do I detail the interaction? One can associate different kinds of interaction specification to an interaction. Figure 65 shows an interaction that is associated with the three kinds of specification methods that the IDK supports at the moment.
- Do I have to worry about the warnings?
    - The warnings show some information to the designer in the code generation, but it does not mean there is something wrong. Indeed, almost all of our MAS specifications produce warnings in the generation, but it is nothing wrong about them. In addition,the warnings will not stop the code generation. Thus, do not worry about the warnings, just consider them.
- Why the code generation is not writing anything?

- o The most likely reason is that the user has changed the name of the project folder in the workspace. In this case the generation of code is not writing where the user expects. For fixing this problem, just press in the "Project" menu, the options properties. There, one should change the expression "../workspace/old_name" with the expression "../workspace/new_name" in the paths. Then, save the project and regenerate.

- How can I learn to use INGENIAS?
  - o This document is a good first step in learning INGENIAS. However, practitioners can find more reference material (such as, the INGENIAS Agent Framework manual) and more examples in http://grasia.fdi.ucm.es (in "Training" section of the web).

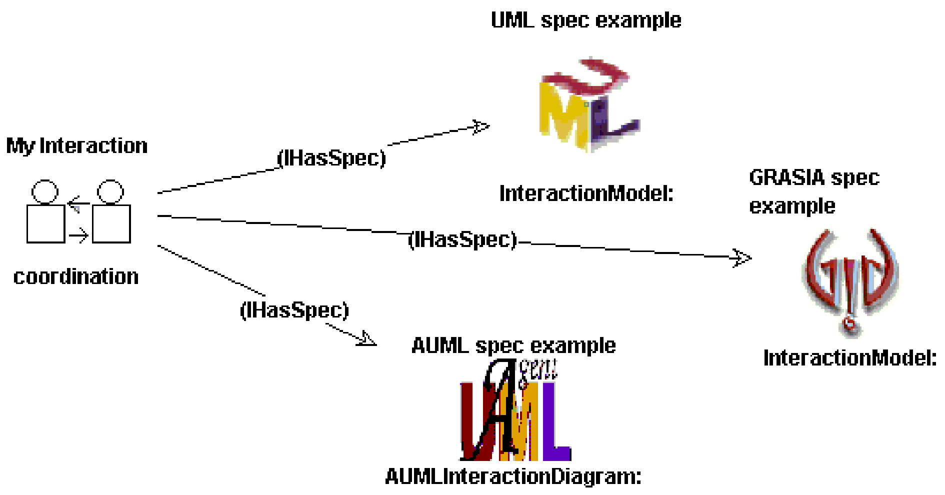


Figure 65. Interaction protocol specification by means of different specification mechanisms

7. A Full Development Example: Crisis Management

This section presents the full development of an example of MAS, which solves the case study of the crisis management. In this case study, a poisonous material has been accidentally released into a city. The number of affected people is very high to be managed only with a centralised solution. The official medical services are not enough to heal all the affected people. Thus, a distributed solution is necessary. The collaboration among the people on the ground is also necessary.

This document presents a MAS for solving the selected case study. The presented solution is based on the following facts. The citizens with medical capabilities can help affected citizens who are close enough. The citizens can be quickly warned of the infected locations to avoid them. The central official system must be informed of all the infected locations. The communications should be efficient. In conclusion, the three goals of the presented MAS are the following, coordination among citizens, network efficiency and information of central services. The whole specification and source code is available at http://grasia.fdi.ucm.es (in "Training"-> "Full Development Examples" -> "Crisis Management")

The full development example is presented as follows: firstly, the requirements of the MAS are indicated; secondly, the main design decisions are described; finally, the diagrams of the specification are presented.

7.1. Requirements

The requirements of the case study are formalised with use cases in the *MainUseCases* diagram (see Figure 66). The first use case is called *CoordinationForHelpingEachOtherUC*. The goal of this use case is the coordination among the people on the ground. The people on the ground with medical capabilities offer help. The poisonous-affected people ask for help. The goal of the mentioned use case is coordinating them to make it possible the people on the ground can help each other.

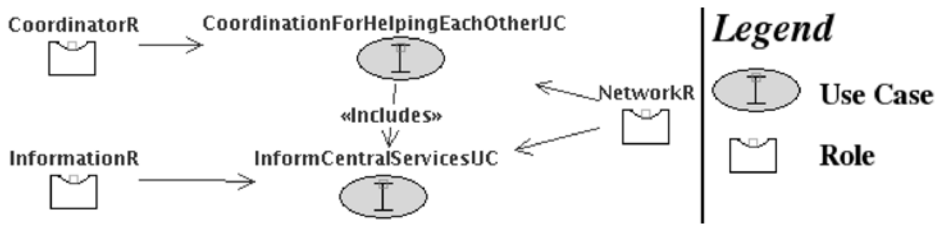


Figure 66: MainUseCases Diagram

The second use case is called *InformCentralServicesUC*. The goal of this use case is to inform the central services of the poisonous-affected locations. The InformCentralServicesUC use case is included in the CoordinationForHelpingEachOtherUC for the following reason. The coordination among people on the ground includes to report the infected locations to the central services. This makes it possible that the central services can assist areas with many affected locations.

Both mentioned use cases employ the network for communication. The network is important to make the communication efficient. Thus, a participant for managing the network is included in both use cases. The network-participant is the intermediator in the communications. The goal of this participant is to improve the efficiency of the network.

7.2. Solution

The presented MAS satisfies the requirements mentioned at previous section . The kind of agents used in the MAS are *coordination-agents, network-agents* and *information-agent* . The *coordination-agents* are responsible for the coordination among the people on the ground. Each mobile device of the people on the ground is habited by a coordination-agent. Each user interacts with a coordination-agent. The *network-agents* are in charge of the proper communication among other kind of agents. The *information-agent* organises the infected locations and shows them to the central official services.

7.2.1 Main design decisions

This design decisions are classified in several groups within this section.

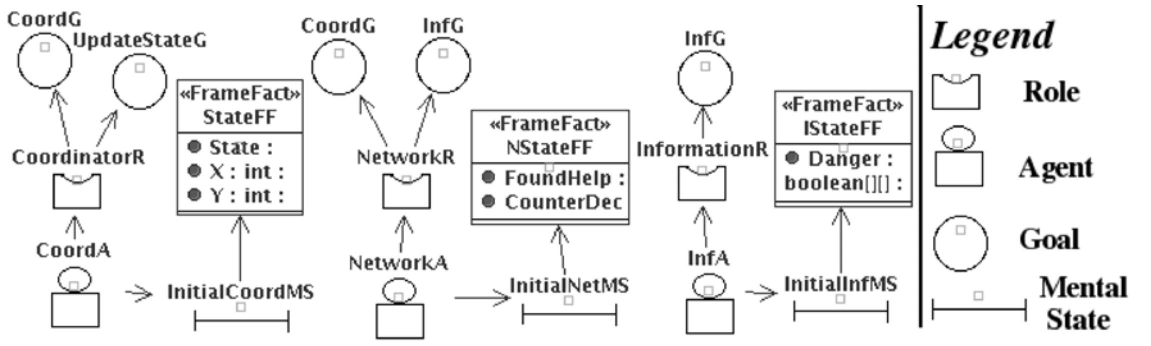


Figure 67: Role and Agent Definitions of the Presented MAS

7.2.1.1 Roles and goals

The presented MAS contains several kinds of agent. Each kind of agent (see Figure 35) is represented with a role. The roles of the presented MAS are the following.

- *Coordinator Role*. Several agents play this role. The goal of the coordination-agents is to coordinate the people on the ground. This coordination makes the people on the ground help each other. It also makes them warn each other of the poisonous-affected locations. Each user can interact with a coordination-agent.
- *Network Role*. Several agents play this role. The network-agents are the intermediators for the communications in the presented MAS. The reason is to isolate the possible communication problems in this role. The goal of this role is to improve the network efficiency.
- *Information Role*. Only one agent plays this role. The goal of the information-agent is to inform of the infected locations. The information-agent contains a city map with the infected locations.

7.2.1.2 Mental States

The mental state of a coordination-agent contains the state of the corresponding user. The user state contains the user location in the city. The user state also contains one of the following values: *need-help*, *can-help*, *being-helped* y *helping*. The user state is necessary for the proper cooperation among the agents.

The mental state of the information-agent contains a city map with the poisonous-affected locations.

The network-agent mental states only contains the facts necessary for the communications. The initial mental states are shown at Figure 67.

7.2.1.3 Interactions

The communication in INGENIAS is defined with interactions. The overall process of the communication is the following. The user can ask for help in case of infection. The user can also offer help if the user can heal the injuries caused by the poisonous material. In case of infection, the coordination-agent interacts with the network-agent with the *Coordinator-networker* interaction.

In case of receiving a message of infection, the network-agent is responsible for three operations. Firstly, the network-agent must look for help around the infected location, among other citizens, through the coordination-agents. Secondly, the network-agent must warm other citizens through the coordination-agents. Finally, the network-agent must report the infection location to the information-agent. The two first operations are achieved with the *Networker-coordinator* interaction. The third one is achieved with the *Networker-informer* interaction.

The most relevant communication decisions are the following. In all the communications, a network-agent is the intermediate. The reason is the following. The problems of the network can be solved with the network-agents regardless other problems. In the future, if network problems occur, the only agents to be changed are the network-agents. Thus, the communication between people on the ground involves two interactions, a Coordinator-Networker interaction, and a Networker-Coordination interaction. The communication between a person on the ground and the information central system involves two interactions, the Coordinator-Networker interaction and the Networker-Information interaction.

One communication among citizens satisfies two necessities. The first necessity is to ask for help around the affected location. The second necessity is to warn other citizens of the affected location. A citizen just reports the necessity of help because of the poisonous. This message is delivered to other citizens. Two cases are possible. The receiver can help the requester. In this case, the receiver coordination-agent ask its user to go to assist the requester citizen. Otherwise, the receiver cannot help the infected person. In this case, the receiver coordination-agent warns its user of the affected location. Thus, the same interactions are used for both warning and asking for help.

A second option would be to use different interactions for each necessity. The selected option reduces the number of interactions above the second option. This fact increases the efficiency of the network.

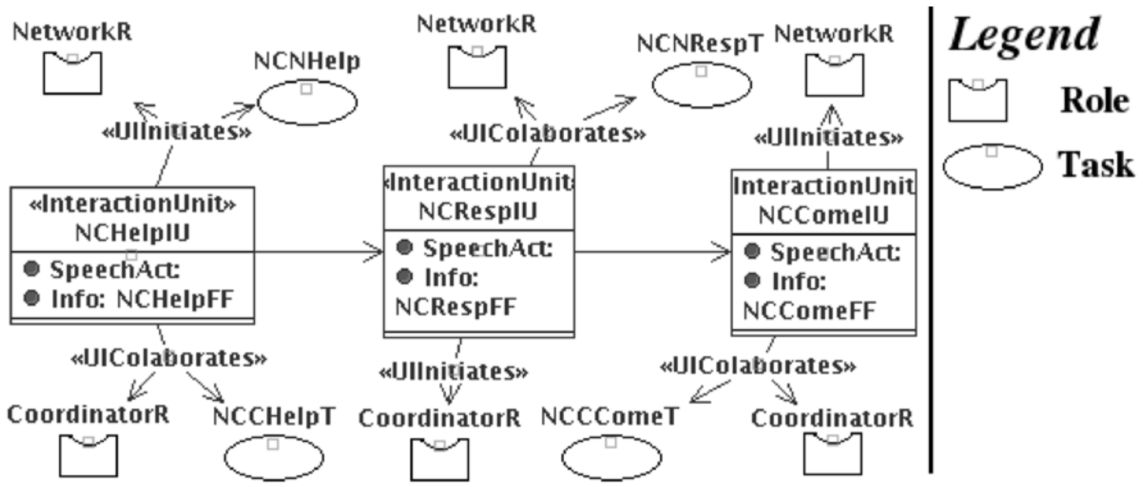


Figure 68: Networker-Coordinator Protocol. A network-agent searches for help. The networker sends a message to the coordinator-agents. The coordination-agents answer indicating whether they can help or not. If possible, the networker selects one coordinator. The networker asks this coordinator to come.

In the search for help, it is possible there are several people on the ground that can assist the same affected person. In this case, the network-agent only asks one of the coordination-agents to come. The reason is the following. Only one helper is needed. The other people on the ground with medical capabilities may be necessary for other affected people. Thus, in the protocol (see Figure 68), firstly the networker asks for help to all the coordinators. Then, all the coordinators answer whether they can help or not. Finally, the networker asks to come only one coordinator.

An interaction is shared for the two necessary communications. First communication is established among people on the ground. Second communication is established from a person on the ground to the central information system. Both communications share the first interaction, called Coordinator-networker interaction. This interaction is initiated from a coordination-agent. This agent sends a message asking for help to the network-agent. Then, the information-agent initiates two interactions. One interaction completes the communication between people on the ground. The other interaction completes the communication from person on the ground to the central-system.

The advantages of sharing this interaction are the following. Less interactions take place in the MAS for the same functionality. Less interactions implies more network efficiency.

7.2.1.4 Participants of the Interactions

The INGENIAS Agent Framework(IAF) generates a default behaviour for one-to-one and one-to-many interactions. In each case, by default, some participants are added to each kind of interaction. However, this default behaviour is not always the most appropriate.

When a coordinator-agent asks for help, this agent establishes the interaction with one network-agent. Thus, this interaction is one-to-one. By default, the first network-agent is always selected. However, this default behaviour is not the most efficient for the following reason. The first network-agent can get collapsed while the other network-agents are just doing nothing. The policy selected for the presented MAS is the following. In the deployment specification, the same number of coordination-agents and network-agents are instantiated. Each coordination-agent is associated with a different network-agent. Each coordination-agent interacts with its associated network-agent when asking for help. This policy increases the reliability and the efficiency of the network for the following reason. The first network-agent does not get collapsed as in the default behaviour. If the coordination-agent cannot contact the associated network-agent, the coordination-agent interacts with another network-agent selected randomly.

A network-agent can search for a coordination-agent that can help a requester coordination-agent. In this search, the network-agent initiates a one-to-many interaction. The default behaviour is to add all the coordination-agents to the interaction. However, the default behaviour is not the most appropriate for the following reason. The requester coordination-agent is also asked for help. The requester should not receive its own message. It is not necessary. The unnecessary messages make the network efficiency lower. The selected policy for the Networker-coordinator interaction is the following. A network-agent searches for help for a requester coordination-agent. The network-agent initiates a conversation with all the coordination-agents but the requester. In this manner, the unnecessary messages or unnecessary collaborators in the interactions are avoided. Thus, the efficiency is increased above the default behaviour.

7.2.1.5 Tasks

The tasks of the presented MAS can be classified into the several groups. A group of tasks is the responsible for reacting to the events generated by the people on the ground. A person on the ground can generate several events from the Graphical User Interface(GUI) provided by the corresponding coordination-agent. These events are *need-help*, *can-help*, *update-location* or *none*. For all the mentioned events, the corresponding task consumes the event. The task updates the user state in the coordination-agent mental state. In particular, for the *need-help* event, the corresponding task launches a Coordinator-networker conversation to ask for help.

Another group of tasks is associated to the *Coordinator-networker* interaction. These tasks transfer the information of the requester (the agent ID, and its location). The most relevant task is the *CNNHelpT* task. This task receives a call for help. This task is executed by the network-agent. This task launches two different conversations. The goal of the first conversation is to search for help from other coordination-agent. The goal of the second conversation is to inform the information-agent of the new affected location. In addition, within the coordination-network interaction, some tasks are the responsible for bringing back the response to the requester agent. The response indicates whether help is found. If help is found, the response contains the name of the coordination-agent whose user is coming to help the poisonous-affected user.

Another group of tasks is associated to the *network-coordinator* interaction. This interaction is related to the search of help. Firstly, the network-agent delivers a frame fact, called *NCHelpFF*, requesting help to all the coordination-agents. This NCHelpFF frame fact contains the requester agent ID and its location. The *NCCHelpT* task is associated with the reception of the mentioned frame fact. This task checks out whether the corresponding user can heal the poisonous material effects, by consulting the coordination-agent mental state. Then, if the user has medical capability, the distance between the the requester user and the helper user is calculated. If both users are close enough, the response is positive. Otherwise, the response is negative. Another task executed by the network-agent, manages all the responses. At most, this task only requests one user to come for help.

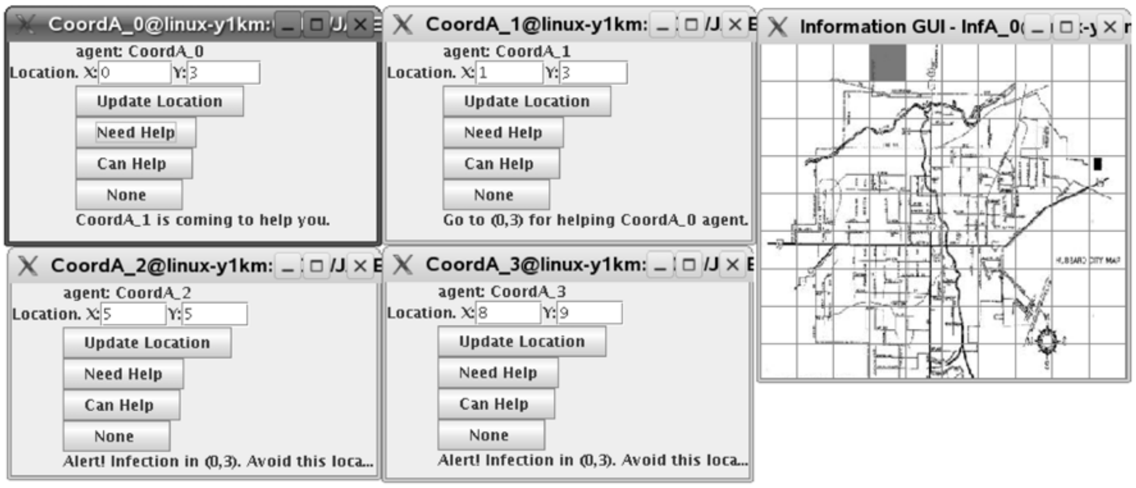
Finally, some tasks are the responsible for the transfer to the information-agent of the new affected location. One of these tasks updates the city map of the affected locations in the information-agent mental state. This city map is shown with a GUI.

7.2.2 Implementation

The presented MAS implementation is available at http://grasia.fdi.ucm.es (in "Training"-> "Full Development Examples" -> "Crisis Management"). The implementation skeleton is generated with the INGENIAS Agent Framework (IAF) code generator. Then, the skeleton is completed manually with programming code. The MAS runs on the JADE platform.

An example of execution is shown in Figure 37 In this example, four people on the ground use the presented MAS. The CoordA_1 and CoordA_3 users have medical capabilities. Thus, both of them press the *Can Help* button. The CoordA_0 user gets affected by the poisonous material. Thus, the CoordA_0 presses the *Need Help* button. The MAS starts to coordinate and inform the central system. CoordA_1 can help CoordA_0 because of two conditions. CoordA_1 has medical capabilities and is close enough to CoordA_0 (look at the locations in Figure 69). On the contrary, no other coordination-agent satisfies the same conditions.

The response is sent back to CoordA_0 with the following message, ``CoordA\_1 is coming to help you''. The other people on the ground are warned of the new poisonous-affected location with the following message, ``Alert! Infection in (0,3). Avoid this location''. Finally, the city map of the central services is updated. The new affected location (0,3) is indicated in the information GUI with a different background colour.



agent: CoordA_0
Location. X: 0 Y: 3
CoordA_1 is coming to help you.

agent: CoordA_1
Location. X: 1 Y: 3
Go to (0,3) for helping CoordA_0 agent.

agent: CoordA_2
Location. X: 5 Y: 5
Alert! Infection in (0,3). Avoid this loca...

agent: CoordA_3
Location. X: 8 Y: 9
Alert! Infection in (0,3). Avoid this loca...

Figure 69: Example of execution.

7.2.3 Testing

The presented MAS includes a battery of tests (see the next section). These tests are modelled with INGENIAS methodology.

The first test is called *GeneralTest* . This test checks out several functionalities of the MAS. The MAS is initialised with the same values of the execution example (see Figure 69). In this case, a citizen requests for help. There are other three citizens. One of them is too far for helping the requester. Second one has not medical capabilities. Finally, the third one is close enough and has medical capabilities. This test checks out the following fact. A citizen is asked to come for helping when the citizen satisfy two conditions. Firstly, the citizen must be close enough to the requester. Secondly, the citizen must have medical capabilities. Moreover, the GeneralTest also checks out the information of the central services. The city map of the poisonous-affected locations is checked at the end of the execution.

The second test is called *OnlyOneComesTest*. This test is initialised with several people on the ground with medical capabilities. The used deployment is called *SeveralDoctors*. An agent asks for help. There are several citizens with medical capabilities around. Thus, there are several citizen that satisfy the necessary conditions for helping the requester. However, only one citizen must be asked to go to assist the requester for the following reason. Only one doctor is necessary. The other doctors must be left available for future requests. This test checks the mental state of the doctors at the end. The test checks out that one and only one doctor is helping the requester.

Therefore, the provided tests can check the main features of the presented MAS.

7.3. Diagrams of the Specification

The whole specification and the necessary source code are available at http://grasia.fdi.ucm.es (in "Training"-> "Full Development Examples" -> "Crisis Management"). From the specification and the source code, the whole system can be regenerated with the IDK2.8 and IAF code generator. The specification can be opened from IDK2.8. The IAF code generator can regenerate the code. For further instructions, consult the "Ingenias Agent Framework Development Guide" in AF Manual at http://grasia.fdi.ucm.es (in "Training").

The MAS can be executed in stand alone version with "ant runSimpleProdStandAlone" command or any of the *start* scripts. Otherwise, two commands must be executed in two different consoles. These two commands are the following, "ant runjadeSimple" and "ant runSimple". In this case, one can observe the jade platform and the INGENIAS console. In this manner, the execution of the MAS can be monitored.

The whole specification is located in "spec/specification.xml" within the source code available. This specification can be loaded with the IDK2.8 specification editor. However, the most relevant parts of the specification are shown in this section.

The specification of the agents is already shown at Figure 35. A group of tasks is responsible for perceiving the people on the ground events and reacting. This group of tasks are specified in *StateTasks* diagram (see Figure 70).

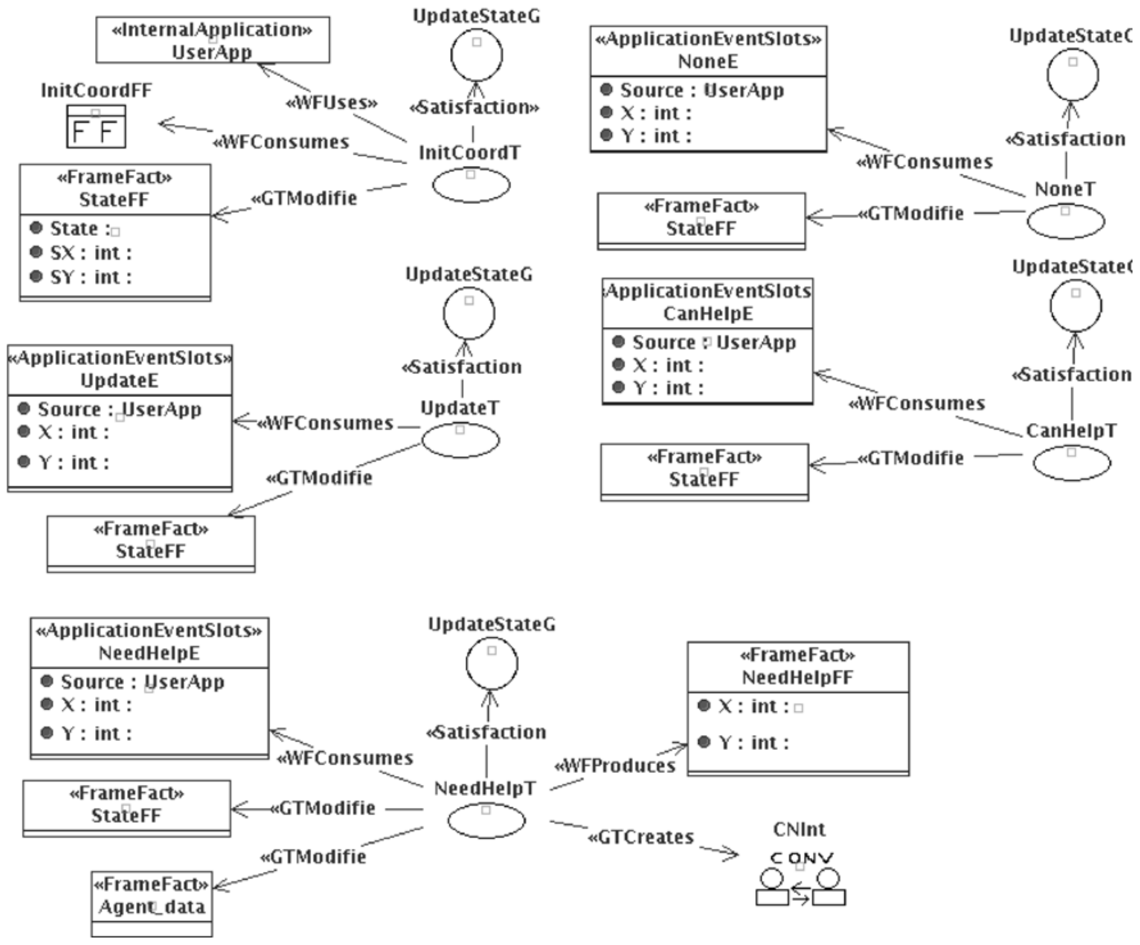


Figure 70: StateTasks Diagram. This diagram contains tasks for perceiving the user events. All these tasks update the state of the corresponding coordination-agent and its location. The NeedHelpE event has further effects. This event launches a coordinator-networker conversation.

The *CNInt* interaction is established from the a coordinator-agent to a network-agent. The CNInt interaction is defined with the *CNDef* diagram (Figure 71). The protocol of this interaction is specified with the *CNProtocol* diagram (Figure 72). The tasks related to the CNInt interaction are defined within the *CNTasks* diagram (Figure 73).

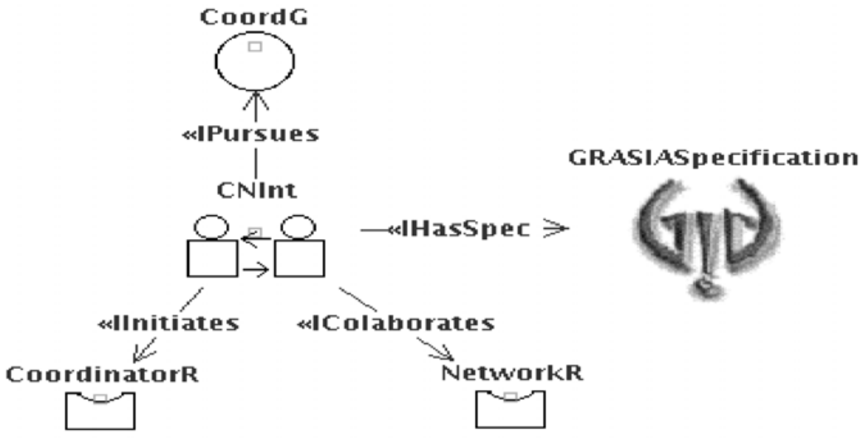


Figure 71: CNDef Diagram. Definition of the Coordinator-networker Interaction.

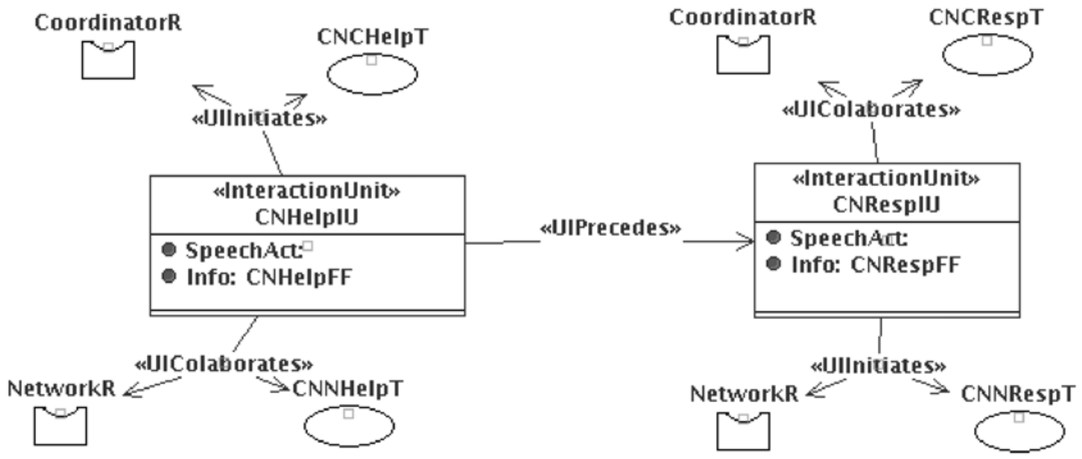


Figure 72: CNProtocol Diagram. Protocol of the Coordinator-networker Interaction. The coordinator-agent asks for help. The network-agent searches for another coordination agent that can help the requester. After this search, the network-agent responses to the current coordinator-agent with the search results.

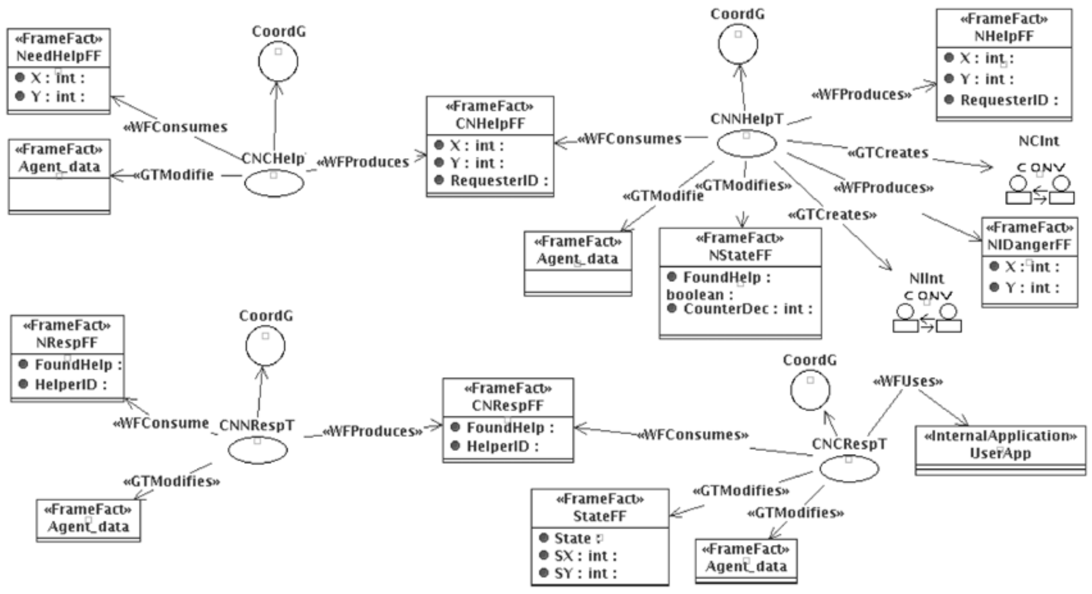


Figure 73: CNTasks Diagram. Tasks Related to the Coordinator-networker Interaction. When the network-agent receives the help message, two conversations are launched. The first conversation searches for help among other coordination-agents. The second conversation informs the central services. The response to the current coordination agent has the following information. Firstly, the response includes whether help is found or not. If so, the helper identifier is also included.

The *NCInt* is established from a network-agent to several coordination-agents. This interaction searches for help from people on the ground and alerts people on the ground. The *NCDef* diagram (Figure 74) defines the interaction. The *NCProtocol* diagram (already shown in Figure 68) determines its protocol. The *NCTasks* diagram (Figure 75) contains the tasks associated to the NCInt interaction.

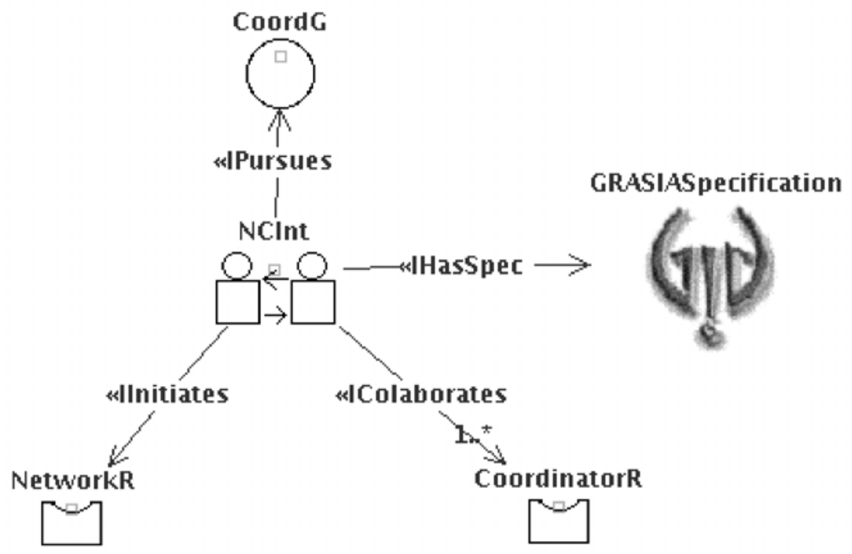


Figure 74: NCDef Diagram. Definition of the Networker-coordinator Interaction. Notice the following fact. This interaction is different from the interaction of Figure 39.

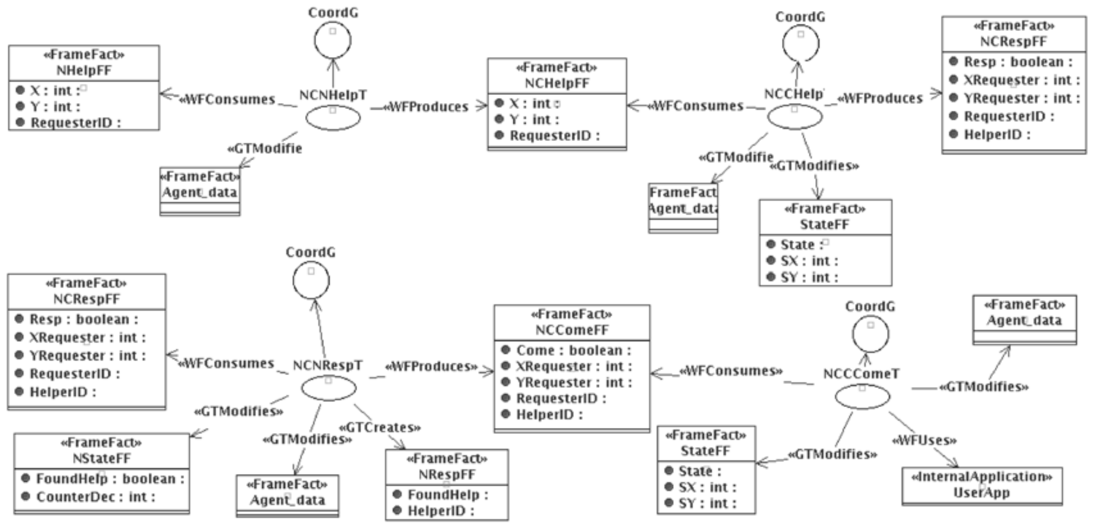


Figure 75: NCTasks Diagram. Tasks Related to the Networker-coordinator Interaction. These tasks transfer the necessary information. This information is the location of the requester, the requester identifier and the helper identifier. The NCCHelpT task calculates the distance from the requester to the current coordination-agent. This task checks the current coordination-agent state. According to these operations, this task answers to the request. The NCNRespT task continues the conversation with the requester communicating the search results by means of the NRespFF frame fact. The NCCComeT task shows a message in the user application depending on whether the current coordination agent is asked to come or not.

The *NIInt* is established from a network-agent to the information-agent. This interaction informs the central service of a new affected location. The *NIDef* diagram (Figure 76) defines the interaction. The *NIProtocol* diagram (Figure 77) determines its protocol. The *NITasks* diagram (Figure 78) contains the tasks associated to the NIInt interaction.

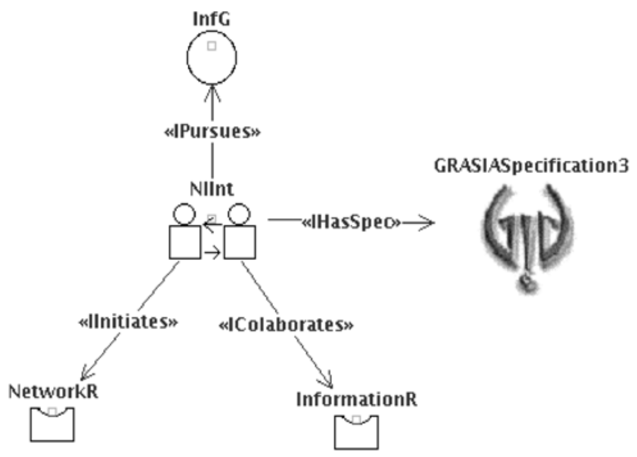


Figure 76: NIDef Diagram. Definition of the Networker-informer Interaction.

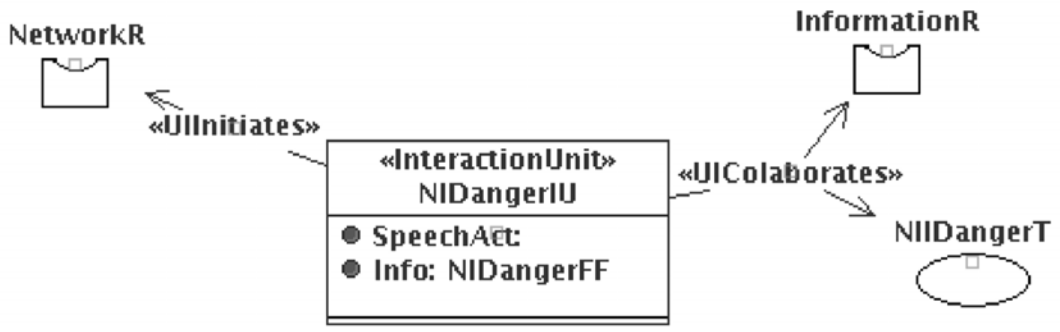


Figure 77: NIProtocol Diagram. Protocol of the Networker-informer Interaction. The networker indicates the new infected location to the information-agent.

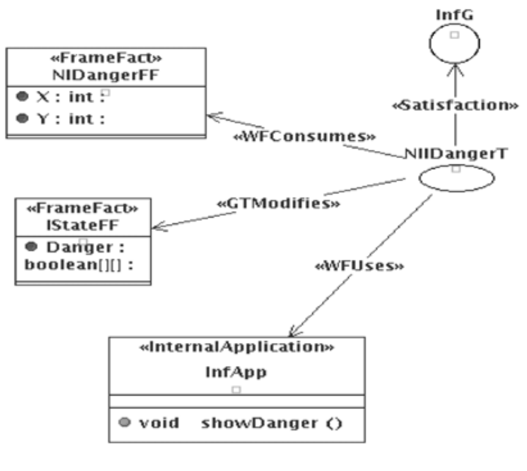


Figure 78: NITasks Diagram. Task Related to the Networker-informer Interaction. The NIDangerT task updates the city map of the poisonous-affected locations.

All the tasks have associated some code in the specification. In addition, the internal applications have some code for initialisation. The definition of the necessary components for the code are omitted in this section, but it it is available in the release code of “Crisis Management”

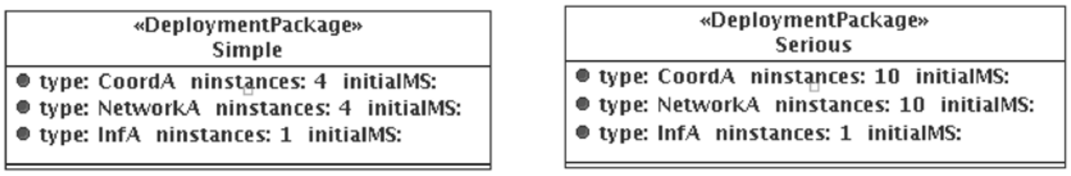

«DeploymentPackage»
Simple
type: CoordA ninstances: 4 initialMS:
type: NetworkA ninstances: 4 initialMS:
type: InfA ninstances: 1 initialMS:
«DeploymentPackage»
Serious
type: CoordA ninstances: 10 initialMS:
type: NetworkA ninstances: 10 initialMS:
type: InfA ninstances: 1 initialMS:

Figure 79: Deployment Diagram. It contains two deployments called Simple and Serious respectively. The Simple deployment initialises four coordination-agents, four network-agents and one information-agent. The Serious deployment initialises ten coordination-agents, ten network-agents and one information-agent.

Finally, the *deployments* diagram (see Figure 79) indicates how many agents are instantiated. There are several deployments for different numbers of users.

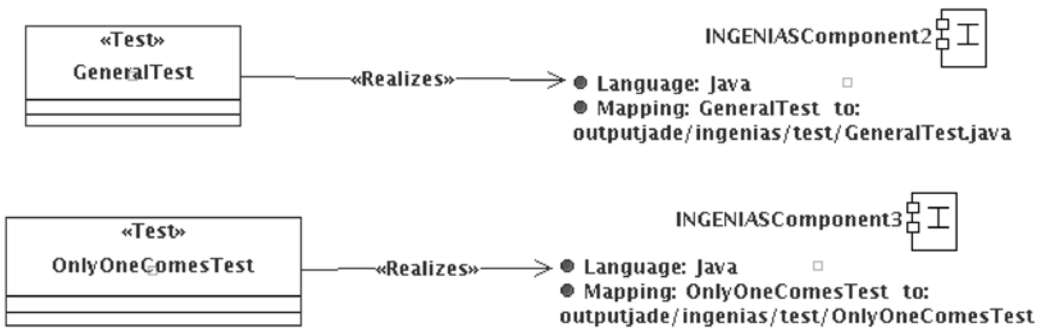


Figure 80: Components for the tests. The test names are GeneralTest and OnlyOneComesTest.

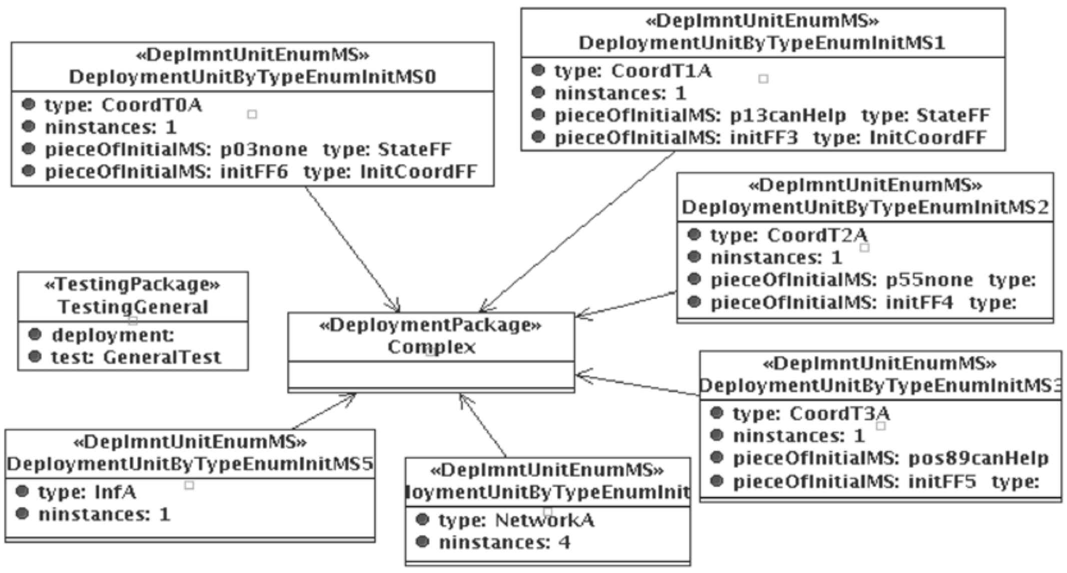

«DeplmntUnitEnumMS»
DeploymentUnitByTypeEnumInitMS0
type: CoordT0A
ninstances: 1
pieceOfInitialMS: p03none type: StateFF
pieceOfInitialMS: initFF6 type: InitCoordFF
«DeplmntUnitEnumMS»
DeploymentUnitByTypeEnumInitMS1
type: CoordT1A
ninstances: 1
pieceOfInitialMS: p13canHelp type: StateFF
pieceOfInitialMS: initFF3 type: InitCoordFF
«DeplmntUnitEnumMS»
DeploymentUnitByTypeEnumInitMS2
type: CoordT2A
ninstances: 1
pieceOfInitialMS: p55none type:
pieceOfInitialMS: initFF4 type:
«TestingPackage»
TestingGeneral
deployment:
test: GeneralTest
«DeploymentPackage»
Complex
«DeplmntUnitEnumMS»
DeploymentUnitByTypeEnumInitMS3
type: CoordT3A
ninstances: 1
pieceOfInitialMS: pos89canHelp
pieceOfInitialMS: initFF5 type:
«DeplmntUnitEnumMS»
DeploymentUnitByTypeEnumInitMS5
type: InfA
ninstances: 1
«DeplmntUnitEnumMS»
loymentUnitByTypeEnumInit
type: NetworkA
ninstances: 4

Figure 81: Deployment for the GeneralTest test. Four coordinator agents are initialised. Two of them have medical capabilities. They are distributed along the city map.

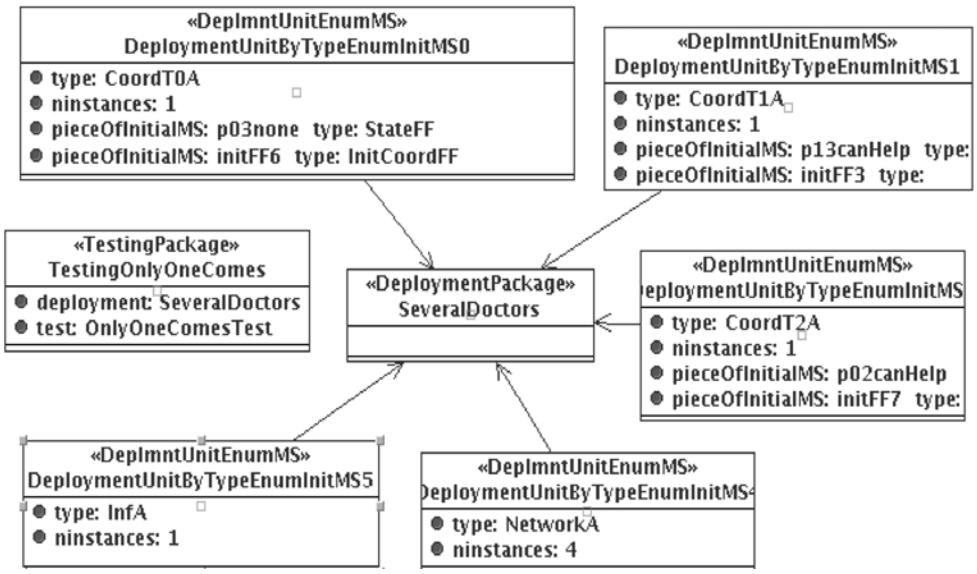

«DeplmntUnitEnumMS»
DeploymentUnitByTypeEnumInitMS0
type: CoordT0A
ninstances: 1
pieceOfInitialMS: p03none type: StateFF
pieceOfInitialMS: initFF6 type: InitCoordFF
«DeplmntUnitEnumMS»
DeploymentUnitByTypeEnumInitMS1
type: CoordT1A
ninstances: 1
pieceOfInitialMS: p13canHelp type:
pieceOfInitialMS: initFF3 type:
«TestingPackage»
TestingOnlyOneComes
deployment: SeveralDoctors
test: OnlyOneComesTest
«DeploymentPackage»
SeveralDoctors
«DeplmntUnitEnumMS»
eploymentUnitByTypeEnumInitMS
type: CoordT2A
ninstances: 1
pieceOfInitialMS: p02canHelp
pieceOfInitialMS: initFF7 type:
«DeplmntUnitEnumMS»
DeploymentUnitByTypeEnumInitMS5
type: InfA
ninstances: 1
«DeplmntUnitEnumMS»
eploymentUnitByTypeEnumInitMS4
type: NetworkA
ninstances: 4

Figure 82: Deployment for the OnlyOneComesTest test. Three coordinator agents are initialized. Most of them have medical capabilities. They are very close to each other.

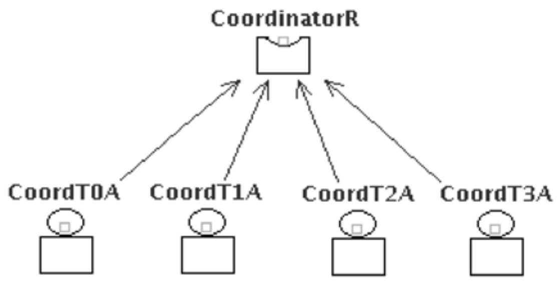


Figure 83: Additional Coordination-agents Defined for Complex Deployments. These deployments are necessary for the testing.

There are two tests. The test components are shown in Figure 80. For these tests, some specific deployments are needed. The deployment of the *GeneralTest* and *OnlyOneComesTest* tests are shown in Figure 81 and Figure 82. Both tests need the user to press the *NeedHelp* button of a certain coordinator-agent to start the MAS cooperation. This fact is indicated properly with a message to the user in the console. For running the tests, two consoles are necessary. In the first console, run the “ant runjade” command. In the second console, run any of the following commands; “ant alljunit”, “ant junitTestingGeneral” or “ant junitTestingOnlyOneComes”.

In the mentioned deployments for the tests, several coordination-agent are initialised with different values. For this reason, several additional coordination-agents are defined to play *CoordinatorR* role. The definition of these additional agents are shown in Figure 83.

References

Apart of this manual, there are other publications on INGENIAS and the IDK. Look at the comments at each reference in order to see whether it can be useful to complete the information and understanding on IDK. Note that some publications are in French or Spanish.

Specific documentation on some IDK modules:

[1] Gómez Sanz, J. manual of IAF

IAF (INGENIAS Agent Framework) is an IDK module that generates code on JADE platform and allows to animate the specification to check whether it behaves as expected.

More documentation about INGENIAS:

[2] J. Pavón, J.L. Pérez de la Cruz (eds.). Agentes Software y Sistemas Multiagente: Conceptos, Arquitecturas y Aplicaciones. Pearson Educación-Prentice Hall. 2005. (in Spanish)

Section 5.7 provides a detailed example on how to drive the design of a MAS, from requirements to implementation, using INGENIAS.

[3] Botía, J.A., Gómez-Sanz, J., Pavón, J. *Intelligent Data Analysis for the Verification of Multi-Agent Systems Interactions.* In: 7th Int. Conference on Intelligent Data Engineering and Automated Learning (IDEAL 2006). E. Corchado et al. (Eds.), LNAI 4224, Springer-Verlag (2006) 1207-1214.

Section 5.7 provides a detailed example on how to drive the design of a MAS, from requirements to implementation, using INGENIAS.

[4] Gómez Sanz, J. *Modelado de Sistemas Multi-Agente*. Ph.D. thesis. Universidad Complutense Madrid. 2002. (in Spanish)

This thesis provides a detailed description of INGENIAS meta-models and the INGENIAS process following a Unified Process approach.

[5] Gómez Sanz, J. & Pavón, J. *Meta-modelling in Agent Oriented Software Engineering*. In: Proc. 8th Ibero-American Conference on AI (Iberamia 2002), F.J. Garijo, J.C. Riquelme, M. Toro (Eds.), Advances in Artificial Intelligence, LNAI 2527, Springer-Verlag (2002) 606-615.

This is one of our first papers about MAS meta-modelling, the basis for the INGENIAS meta-model.

[6] Gómez-Sanz, J. & Pavón. *Implementing Multi-agent Systems Organizations with INGENIAS*. In: Programming Multi-Agent Systems: Third International Workshop, ProMAS 2005, Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, Amal ElFallah Seghrouchni (Eds.), LNCS 3862, Springer-Verlag (2006) 236 - 251.

This paper describes how organizations can be modelled with INGENIAS and how the organization viewpoint relates with other viewpoints.

[7] Pavón, J. *INGENIAS : Développement Dirigé par Modèles des Systèmes Multi-Agents*. Dossier d'Habilitation à Diriger des Recherches de l'Université Pierre et Marie Curie (2006). (in French)

The provides a good description of INGENIAS model driven approach and the use of IDK for French speaking people.

[8] Pavón, J. & Gómez-Sanz, J. *Agent Oriented Software Engineering with INGENIAS*. In: Proc. 3rd International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2003), V. Marik, J. Müller, M. Pechoucek (Eds.), Multi-Agent Systems and Applications II, LNAI 2691, Springer-Verlag (2003) 394-403.

The first paper providing an overview of INGENIAS.

[9] Pavón, J., Gómez-Sanz, J.J., Fuentes, R. *The INGENIAS Methodology and Tools.* In: Agent Oriented Methodologies. B. Henderson-Sellers and P.Giorgini (Eds.) IDEA Group Publishing (2005) 236-276.

This book chapter provides a more detailed description of INGENIAS than . Recommended reading for introducing to INGENIAS.

[10] Pavón, J., Gómez-Sanz, J.J. & Fuentes, R.: Model Driven Development of Multi-Agent Systems. In: European Conference on Model Driven Architecture (2006), LNCS 4066, Springer Verlag (2006) 284-298.

This paper introduces the principles of the INGENIAS approach to model driven development of MAS.

[11] Jorge J. Gómez-Sanz, Rubén Fuentes-Fernández, Juan Pavón, and Iván García-Magariño. INGENIAS Development Kit: a visual multi-agent system development environment (BEST

ACADEMIC DEMO OF AAMAS'08). In The Seventh International Conference on Autonomous Agents and Multiagent Systems, AAMAS'08, pages 1675-1676, 2008. May 12-16, 2008, Estoril Portugal.

This paper introduces of the IDK tool. It presents the advances in 2008 about debugging MAS with the INGENIAS methodology. The academic demo about this paper was selected as the Best Academic Demo of AAMAS'08 conference. The demo presented the tool with the example of a Delphi process in the domain of document relevance.

[12] Iván García-Magariño, Jorge J. Gómez-Sanz, and José R. Pérez Agüera. A Multi-Agent Based Implementation of a Delphi Process. In The Seventh International Conference on Autonomous Agents and Multiagent Systems, AAMAS'08, pages 1543-1546, 2008. May 12-16, 2008, Estoril Portugal.

This paper presents a MAS for evaluating whether a document is relevant or not. This MAS was built from its inception to its deployment with the INGENIAS methodology and the IDK tool. This MAS uses the Delphi process, which is based on rounds of questionnaires that are filled up with expert agents. A moderator agent collects questionnaires and creates other questionnaires inspired in the replies of the expert agents.

[13] Iván García-Magariño, Jorge J. Gómez-Sanz, and Rubén Fuentes-Fernández. INGENIAS Development Assisted with Model Transformation By-Example: A Practical Case. In 7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'09), 2009.

This paper presents a tool, called MTGenerator, for generating model-transformations for MAS. This tool allows one to create model-transformations for INGENIAS modelling language, by just introducing model prototypes. The presented tool is inspired in the model-transformation by-example principles.

Related documentation:

[14] Eclipse documentation, http://www.eclipse.org/documentation/

[15] JADE. Java Agent DEvelopment Framework. http://jade.tilab.com/

[16] JESS. Java Expert System Shell, http://herzberg.ca.sandia.gov/jess/

[17] OMG. Meta Object Facility (MOF) Specification. Version 1.4 (2002) formal/02-04-03.