Grupo *GRASIA!*
http://grasia.fdi.ucm.es

Facultad de Informática
Universidad Complutense de Madrid
28040 Madrid
España

INGENIAS Agent Framework

Development Guide version 1.0

AUTHOR: Jorge J. Gómez-Sanz

Introduction

This document introduces to the basics needed to produce specifications with the INGENIAS Development Kit's, or IDK, editor which can be processed by the INGENIAS Agent Framework or IAF. The IAF produces native Java code over the JADE platform. It has its own agent architecture, separated from Jade's behavior framework.

Scope

Readers of this document should be familiar with the INGENIAS Development Kit facilities. Therefore, reading the IDK manual is a must (http://ingenias.sourceforge.net). Other software that should be known by the reader is ant (ant.apache.org) and JADE (http://jade.tilab.com).

It is recommended to be familiar with agent theory, in particular with the concepts of Organization, Believes Desires Intention model, agent coordination, and mental state.

This document should be read by any one interested in covering the implementation stage of an agent oriented development with INGENIAS.

INDEX

1 What is the IAF

IAF stands for the INGENIAS Agent Framework. It is a framework developed along several years that enables a full model driven development. This means that a developer can focus most of its effort in specifying the system, converting a great deal of the implementation in a matter of transforming automatically the specification into code.

This IAF permits to combine the classic approach for coding applications with modern techniques of automatic code generation. The resulting system is almost fully operational, reducing the amount of work of the developer in an relevant degree.

Each produced MAS works over the JADE platform. Hence, additional tools existing for this framework can be applied as well.

The most important aspect of the IAF is its demostrated capability to solve problems. It has been applied along several projects with very promising results.

Preliminar versions of this module has been released in the past, though this is the first time a complete development manual is produced as well.

1.1 IAF and the IDK

The IDK is the specification editor that hosts plugins for the parsing of the current opened specification project.

Both IDK and IAF can work together or separatedly. The IAF has its own folder, as well as the IDK. The IAF can be executed from the command line or invoked from the IDK directly.

The IDK includes by default a module copy of the IAF and enough infrastructure to perform the instructions included in this manual. Nevertheless, readers are invited to get used to working with the IAF alone.

1.2 Location Conventions

To properly understand the instructions to develop a project, it is assumed that the IDK and the IAF will be allocated as follows:

- **IDK** folder. It hosts the INGENIAS Development Kit..
  - **IAF** folder. It contains the INGENIAS Agent Framework, including binaries and sources.
  - **workspace** folder. It contains zero or more **Project** folders.
    - A **Project** folder contains the current project under development.
  - **editor** folder. The visual editor is allocated here.

2 A MAS according to the IAF

A MAS in the IAF is constructed over the JADE platform. The MAS can be distributed along one or several containers in one or many computers. To enable this feature, the IAF has means of declaring different deployment configurations.

The running MAS will be connected to several non-agent applications providing the basic services. Hence, if the MAS has to interact with a user, there will be GUIs producing events according to user actions, and defining actuators for agents. These GUIs will be specified as applications at the specification level. These aspects are considered in the section about application integration.

An important feature of the IAF is the relevance of interactions, which are considered first class citizens during specification and coding. An interaction in runtime is called a conversation. The interactions according to the IAF have the main purpose of transferrring information from one agent to another. This information transfer is ruled by timeouts and initiation/colaboration conditions. Also, interactions can be aborted due to failures in the communication or simply because an agent did not answer within the timeout. Finally, interactions consider cases where there may be several actors of the same time, i.e., supports broadcasting of information.

Tasks are important as well. They are scheduled because the agent wants to attain a pursued goal. The specification of tasks permits them to influence in the mental state by removing/adding information, start conversations with other agents, or influence into already running conversations. Tasks support cardinality attributes associated to the inputs.

Testing is a new addition to the IAF. A developer can define at the specification level what tests will be performed and to what configuration of MAS will be applied. The detailed definition of the test has to be handcrafted, though there are some utilities that could make this work easier.

2.1 The lifecycle of an IAF agent

An agent in the IAF performs a simple deliberation cycle:

- Identify new tasks to schedule and tasks to remove from the schedule. The schedule of the agent is a queue. Therefore, those tasks scheduled first are executed first. Tasks to be scheduled are those whose execution can satisfy the pursued goals and whose inputs are available. Due to the execution of some tasks, a scheduled task may have one or more of its inputs removed. Therefore, it can no more be executed. In this case, the task is removed from the schedule.

- Execute one scheduled task. Task execution is not concurrent. This permits to define a safe mechanism to detect modifications in the mental state of the agent that can affect scheduled tasks.

This cycle omits the classic perception step. It is not needed at all. In fact, the agent can receive new information at any moment and this does not cause internal conflicts. The incorporation of new pieces of information does not imply a change in already scheduled tasks, though it may mean the incorporation of new tasks.

Unfortunately, this quality prevents aborting scheduled tasks if new contradicting information appears. Despite this quality, the architecture would still capable of aborting already scheduled tasks, if the developer needs so. For this, it is enough to

remove manually those mental entities existing in the mental state that activated the scheduled tasks.

3 An Automatically Generated Code Development Philosophy

The INGENIAS Agent Framework Framework, or IAF, bases on a philosophy for the automatically generated code generation approach. In this approach, there is a specification which is the core of the development. From this core, the different scripts and sources are automatically produced.

Nevertheless, the produced code is imperfect. It misses some functionaly related to the different concrete processes to be executed within the individual agents of the system. Therefore, it is required to manually embedd this missing code. Also, a developer may find necessary to modify already generated code.

The IAF proposes several means to handle these needs:

- The developer can modify manually the different templates if it is necessary. Such change would affect all files produced from the same template file. Templates are stored within the IAF folder, concretely under the **IAF/src/templates** folder.

- It proposes to allocate part of the code in the specification. It proposes a UML component like solution to associate code to the main elements of the specification.

- It distinguishes three kind of folders, where the manually generated code is, where the generated code is, and where the manually modified code of generated code is.

- It helps with an existing plugin, the AppLinker, to upload automatically API changes in certain Java classes back into the specification

Developing with code generation facilities requires some discipline. Ideally, it will require from the developer the following simplified development cycle:

1. Creating the project in the folder **Project**

2. Creating the specification and storing it in the **Project/spec** folder of the project

3. Modifying the project properties so that they point at the right places of the hard disk

4. Fill in the specification with the MAS definition solving your problem.

5. Generating code from the specification. This step will imply modifying the specification to provide missing information the code generator requires.

6. Probably, adding additional code in the **Project/src** folder of the project

7. Probaly, modifying some generated file sin the **Project/permsrc** folder. These modifications should not delete or modify automatically generated functions, unless in the cases pointed out along this report.

8. Compiling and executing the code.

9. Test the code

10. If necessary, go back to the 4.

This guideline is simplified at some points, specially at steps four to seven. Generating a specification that the code generator can run requires further steps to consider, for instance. Also, integrating the code under folder **Project/src** with code under the other folders, requires additional knowledge and actions dealing with external application integration, as it will be reviewed in the following sections.

Please, do not take the explanations above as a development process. If there is a need of a serious one, I recommend strongly to visit the INGENIAS web page and read how a development process should be.

3.1 Working with the IDK and the code generator

The specification is produced with the IDK, while the code generation is performed with the IAF. The IAF can be launched from the visual editor or from command line. Each one has its advantages and drawbacks:

- Working with the editor and producing the specification for the IDK. It is recommendable initially, so that one can check the appropriatedness of the resulting system against the specification and perform the required changes. This method is fast, since each code generation invocation does not require loading again the specification. Nevertheless, it needs launching the whole editor, an editor that may prevent the MAS execution if the PC has not enough. Usually, the IDK runs well with 256MB. Similarly, a running medium size MAS (like 10 agents) may require 256MB as well, though it depends on the individual complexity of the agents.

- Working with the code generation from command line. It is invoked usually from the project folder. It makes sense when the specification cannot be modified and most of the work is applied to non-automatically generated code. This variant does not require launching the IDK, therefore it saves resources for other purposes. Nevertheless, each time is executed, it requires loading again the specification, so it is a slower method. For a project with multiple developers, this is more adequate, since the generation of the code can be integrated with other project maintance utilities, like Maven.

A problem with these two working methods refers to the allocation of the IDK and the project folder, usually far apart. Having into account that the specification file must contain a referene to the project folder, it is hard to have both the IDK and the command line working properly. The safest option is to use absolute paths, though this is not the most portable solution.

As an alternative, it is recommended: allocate the IDK or the IAF in the same relative path with respect the project. This constraint will affect only those willing to use the IDK visual editor for code generation. For instance, if the IDK was in the folder **/myfolder/foo** and the project was in **/myfolder/bar** for instance, the **JADE main project folder** property will point at **../bar** (see section 4 for more information).

For code generation from the command line, it can be used the same properties stored in the specification. Hence, if the **../bar** value is used, the code generation will still work when launched within the project folder with the following:

```
ant -f generate.xml -Dspecfile=MySpecFile
```

If this constraint is not satisfied and the relative path does not point at the correct folder, the developer can specify directly the project folder from command line.

```
ant -f generate.xml -Dspecfile=MySpecFile
-DmainP=MyProjectFolder
```

4 A Project

First of all, the developer must define a folder where the project will be allocated. From the IAF folder, a developer will execute

```
ant -DmainP=WRITE_HERE_THE_PATH_TO_YOUR_PROJECT_FOLDER create
```

In concrete, is recommended the following path, to keep all projects together:

```
ant -DmainP=workspace/WRITE_HERE_MY_PROJECT_NAME create
```

The project folder points at a point in the file system where the sources and binaries of the project will be stored. The folders created with this command are the following:

- **bin**. Binaries of the project are stored here
- **lib**. Libraries needed by the project. By default, it includes all libraries needed to create a running MAS.
- **doc**. Folder for storing the documentation of the project
- **spec**. Folder containing the current specification. After creation, this folder contains a file named **specification.xml** properly setup to generate correctly code with the IAF.
- **config**. It is the folder containing configuration files. Initially, it contains the Properties.prop file, which is the main configuration file.
- **jade**. It contains files managed by JADE platform.
- **logs**. Here, the developer can find the logs produced by the MAS, if they are enabled.

The project will be executable from a single ant build file, the **build.xml** file. This file is the default *ant* looks for when executed.

There are additional ones which can be configured by the user:

- Folder **gensrc**. The generated source folder contains code that can be regenerated completely from the specification.
- Folder **permsrc**. The permanent source folder that attains those automatically generated sources which are generated only once and never rewritten, unless they are deleted manually.
- Folder **src**. The folder containing sources belonging to the main development, created manually perhaps with a classic development environment.

The previous folders can be configured by the user by modifying the project properties. Project properties are accesible from the main menu of the IDK.

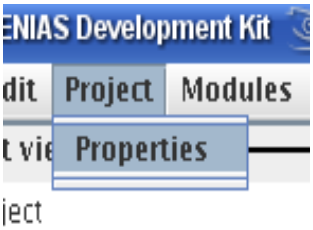

Figure 1: Properties of a project

These properties give access to several properties of the IDK's plugins. For a project, the following plugins make sense:

- IAF code generator. It produces the code files of the project, together with the build.xml files and the default tests. This report is about this code generator mainly.

- AppLinker. It permits to upload changes in an application API to the specification. How the AppLinker works is explained in the IDK documentation.

- HtmlDoc. It generates HTML documentation. How the HtmlDoc works is explained in the IDK documentation.

Each of those plugins have individual sets of properties to configure. The main ones are those from the IAF code generator. The properties editable by the user, which are associated to this module, are the following:

- JADE main project folder. It has to be the same folder as the **project** folder defined previously with *ant* (see the beginning of the section). By default it is **myproject**.

- JADE generated output folder. By default, it is **gensrc**.

- JADE generate only once folder. By default, it is **permsrc**

- Main source folder for the project. By default it is **src**

These properties are related to the default folders presented at the begginning and can be accessed as is it shown in figure 2.

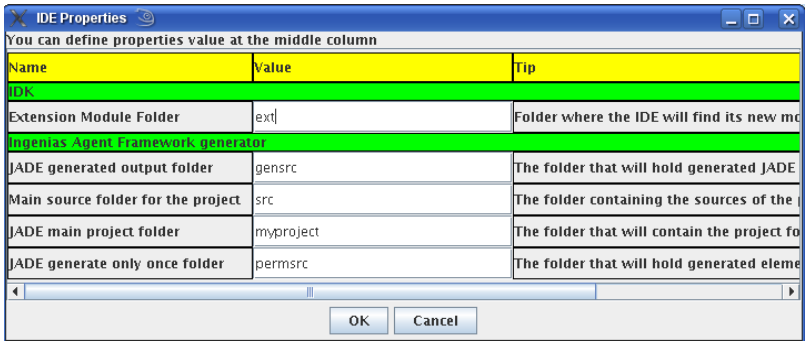

Figure 2: Properties associated to the IAF obtained after following steps from figure 1

These properties are already correctly set if the generated specification.xml file is used as starting point. This file is created with the abovementioned **create** command.

The AppLinker module requires no properties to be setup.

For the pluing HTMLDoc, the unique property to configure the **HTML document folder**. By default, its value is **html**. If the folder is to be stored in the **doc** folder of the project, a developer should change this value with the absolute path to the **doc** folder of the project.

5 Generating the specification

Correctly producing an IAF specification the IAF requires the following aspects to be determined by developers:

- Interactions. They describe how agents do coordinate. Coordination is defined in terms of information transfer units.
- Tasks. They are the basic unit of behavior of an agent. It modifies the agent mental state and performs actions over applications.
- Agents. The main building block. An agent is defined completely when its percepcion, main tasks, and the coordination means it has are described.
- Deployment. It expresses how agents are instantiated and initialized in the final system.
- Tests. Permits to describe testing units the MAS should pass.

6 Mental State

The mental state is composed of a set of goals, conversations, tasks, facts, and events. These entities enable the task execution, therefore, it is important to keep the mental state of the agent as clean as possible.

6.1 Structure of the Mental State

An information entity can be stored in the Mental State directly or in the space defined within a conversation, as it is described in section 7.3. Conversations are entities representing agent interactions in runtime. This serves to isolate the scope of the modifications resulting from a communication and enables an agent to handle several conversations at the same time.

Whenever there is a change in the mental state, developers must take into account that the agent will reconsider its sets of scheduled tasks.

6.2 Mental State Generation

The code generator produces specific Java code for any event or framefact found in a diagram. Goals are represented the same way, through a StateGoal entity, i.e., there are no individual Java classes representing the different goals of the system. This entity adds to a meta-model goal the attribute of state. Conversations are translated in a similar way. For the agent, there will be only RuntimeConversations. These entities

extend a Conversation and provides additional support, like including the information of the state of the conversation as well as error codes, if any.

The generation of code for events and framefacts requires the concrete mental entity to appear explicitly in, at least, one diagram.

The code for each individual frame fact, like the one shown in figure 3, a specific java file is created.

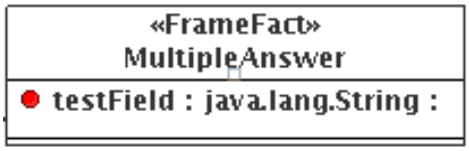

Figure 3: Framefact representation

To define a slot in a frame fact, it is necessary editing the frame fact, going to the slots field, right click in the list, and selecting to create a new one. Never forget to use fully qualified names if the type of the slot is a Java object.

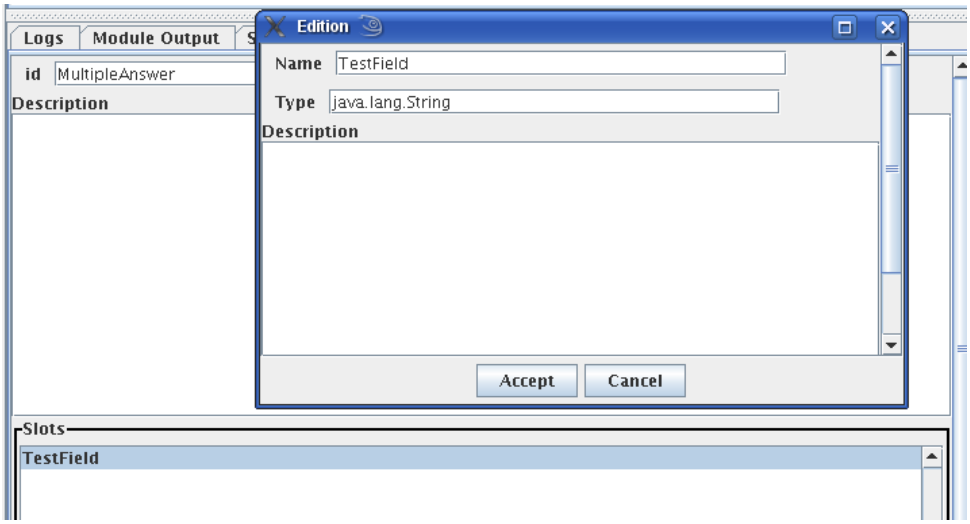

Figure 4: Defining a new slot in the frame fact

The generated file produces an entity extending a RuntimeFact entity. An instance is automatically assigned an id, as well as some useful methods, like the toString and getType. Also, this file contains get/set methods for accessing the different attributes defined. All classes generated are allocated in the **ingenias.jade.mental** package and imported in all tasks by default.

```
package ingenias.jade.mental;
import java.util.*;
import ingenias.jade.components.*;
import ingenias.editor.entities.*;
import ingenias.editor.entities.ViewPreferences.ViewType;

public class MultipleAnswer extends
ingenias.editor.entities.RuntimeFact{

      java.lang.String testField;

      public MultipleAnswer (){
```

```
        super(ingenias.jade.MentalStateManager.generateMentalEntityID());
              this.getPrefs().setView(ViewType.UML);
        }

        public String toString(){
              return this.getId()+":"+this.getType();
        }

        public String getType(){
              return "MultipleAnswer";
        }



        public void settestField(java.lang.String value){
              testField=value;
        };

        public java.lang.String gettestField(){
              return testField;
        }

}
```

Knowing these classess is necessary in order to successfully modify the mental state of the agent. Modification requires calls to set methods, while reading happens through the get methods.

A similar scheme to this one applies as well to events, which in fact uses as well a slot-like definition.

6.3 Mental State during execution

During execution, it is good to monitor the mental state of each individual agent making sure it evolves as it was planned initially. One common mistake is letting the garbage grow too much. If the mental state visualizer of the debugger shows a crowded mental state, probably something is going wrong.

Initially, the agent will start with a mental state made of:

- Mental entities associated to a MentalState entity already associated to the agent. In this case, there will be only empty instances of those entities. Sometimes, this sufficies. If not, the next alternative can be chosen.

- Mental entities defined within a deployment configuration. For more details of this solution, please, refer to section

To help in the mental state maintenance, the IAF includes some facilities:

- A process cleaning all finished or aborted conversations after ten seconds. This amount of time can be configured by the developer with the property ingenias.jade.GarbageCollectionInterval. Read section 15 for more information.

- A default task deleting all entities which are not used by any other task within the agent. This task, in principle, does not conflict with a conversation transmitting this information to another agent, at least if the conversation existed before the task produced the information. To prevent this kind of removal, the IAF defines locks which are released only when the information is transferred. However, if the task produces the information, and then another task produces the conversation expected to transfer the information, there is a high chance of a deletion of the entity to be transferred, since no locks will exist.

6.4 Mental State and Concurrence

Paralelism and concurrence should be a concern in a development with agents. The shared resource to be accessed is the mental state of the agent. The mental state is accessed by the following elements: tasks, interactions, and applications.

Scheduled tasks are executed sequentially always. Nevertheless, the execution order cannot be known by the developer. A developer has to trust in the proper generation of the intermediate mental states which each task is expecting. Nevertheless, the execution of some tasks may altere these states, leading to a situation similar to a write conflict between two processes. It is assumed that the developer knows if two tasks have shared inputs to be consumed. In such case, it is assumed as well that the developer in fact only wants one of those tasks to be executed, which is the implemented behavior. Whether this behavior leads to undesired mental states is something to be considered by the developer.

While a task is scheduled, the mental state which was known by the task may change enough to invalidate the task execution. This situation is controlled by checking the existence of the inputs required by the task right before the execution. Therefore, it can be ensured that whenever a task is executed, it will find the necessary inputs.

Interactions and applications have the effect of adding new information to the mental state. Therefore, their main effect consists in enabling more tasks to be executed. Though applications may delete mental entities from the mental state of the agent, it is highly advised not to do it unless it is strictly necessary.

7 Defining interactions

Interactions are defined into two steps. First the interaction itself is defined. The definition requires inserting an interaction entity, roles, a reference to a protocol definition, and a goal (optionally). Each of those elements are explained following:

- A role initiating the interaction. It is the initiator and there can be only one. It is linked with a *IInitates* relationship.

- One or many roles colaborating in the interaction. There can be as many as needed. They are linked with a IColaborates relationship. The relationship can be further decorated with a cardinality label. This cardinality can be either **1** or **1..*** (same as **1..n**). The default is **1**. It means how many agents playing the role will be incorporated. Having cardinality **1** stands for including only one agent. In the Figure 5, Participant1 has cardinality **1**, though not indicated explicitly. Also, in the Figure 5, *Participant2* has cardinality **1..***, implying that at least one agent and at most all agents playing that role will be included. This can be further defined by the developer within the task code. A

developer can determine which agents will participate in the interaction. By default, if there are **1..***, all agents playing the role are included.

- Interaction entity for representing the interaction. It will serve to refer to this interchange of information.

- GRASIA protocol. It is a reference to another interaction diagram containing the protocol definition.

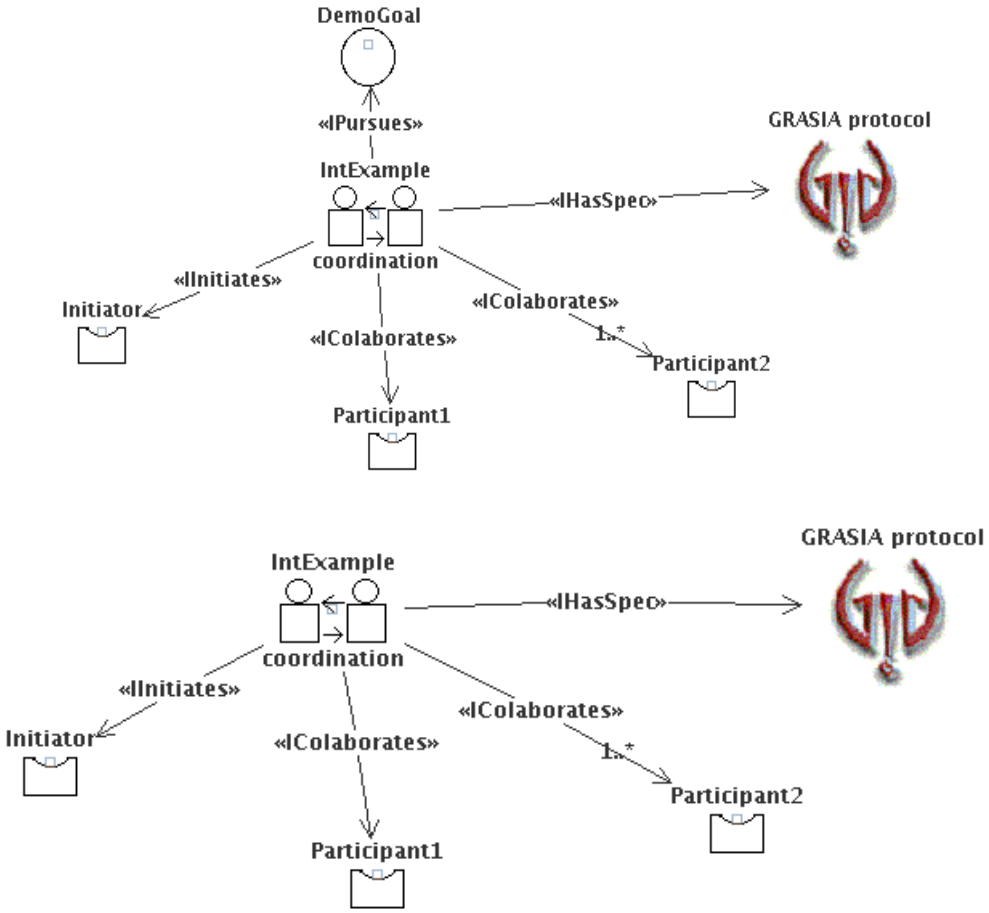

Figure 5.Example of interaction definition

The interaction specifciation entity has to be associated to another interaction diagram where the protocol is further detailed, as figure 6 describes.

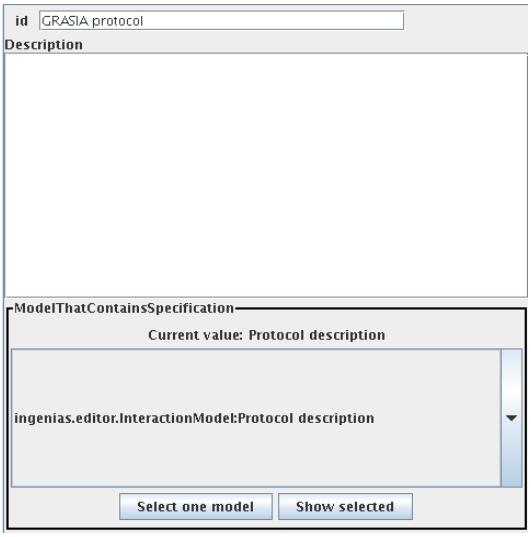

Figure 6: Selection of a protocol

Figure 6 shows the dialog opened for associating the interaction with its protocol. Known interaction diagrams are shown in a combo, so that the developer can choose one. When the one is foind, it is required to press **select one model**.

7.1 Describing a protocol

The interaction protocol is defined with interaction units. An interaction unit represents an information exchange between two agents. These units have an initiator and a colaborator. Initiating means that the role **initiator** will deliver the information, while colaborating means the role **Participant1** or **Participant2** agree to receive the information. The information to transfer from sender to receiver is detailed in the interaction unit. It can be one or several mental entities.

These interaction units are unordered when they are associated to their initiator and colaborator. Order among the interaction units is defined by means of *UIPrecedes*. This relationship means precedence in time. So far, it is not possible to define loops or conditionals, but it is not needed either. Therefore, figure 7 determines that first, *initial question* will be transmitted, then *MultipleAnswer*, later on, *SingleQuestion*, and finally *SingleAnswer*.

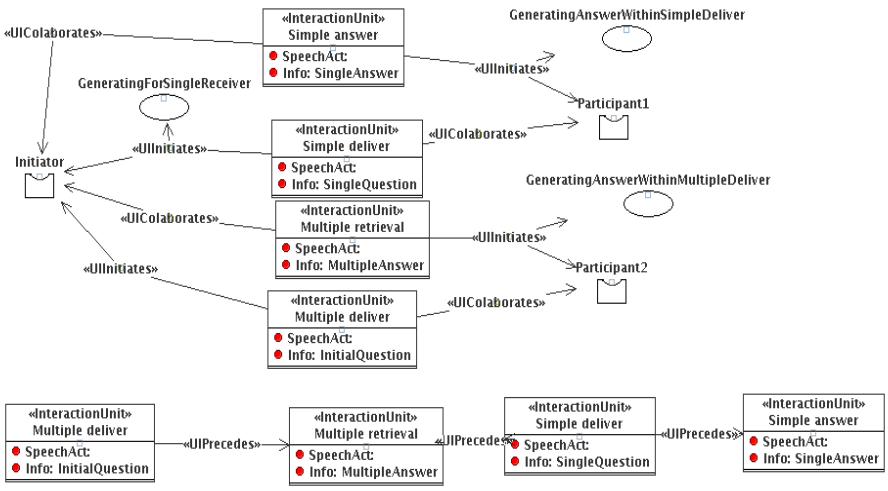

Figure 7: Example of protocol specification

Along the protocol from figure 7, readers may have notified tasks being associated to the *UIInitiates* or *UIColaborates* relationships. These associations are related to how the information used by the interaction is stored in the agent. Briefly, the tasks gain access to the information interchanged during the interaction execution. Review section 7.3 for more information about this aspect.

It is not mandatory to set up interaction units in a linear sequence. In fact, the developer can choose the configuration shown in figure 8. In advance, it must be said that this modification does not make sense according to the meaning of the interaction units in this case. Hence, it should be understood as an example of the effect of different associations. Figure 8 sets two possible courses of action, one is *Multiple deliver*, *Multiple retrieval* and *Simple deliver*. The other one is *Multiple deliver*, *Multiple retrieval* and *Simple answer*. In what respect the interaction, the interaction will in fact try to progress through both lines. Which ever finishes first makes the interaction finish as well. In this case, the interaction will finish with the deliver of the *SingleQuestion*, since *Single answer* interaction unit cannot happen unless *SingleAnswer* fact exists, and this fact will exist only if a *SingleQuestion* fact has been received before.

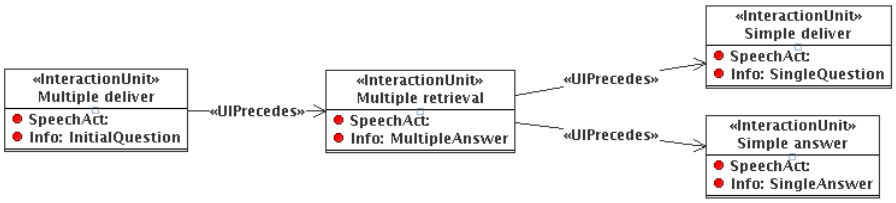

Figure 8: Another arrangement in time of interaction units

7.2 Information transfer

Information transfer happens automatically. Whenever the information is available in the mental state of the sender, the transfer occurs. The information deliver is conditioned by the cardinality of the receiver. If in the interaction definition, the receiver was linked with cardinality **1..***, the deliver becomes a broadcast to all participants playing that role. Similiarly, if the sender appeared with cardinality **1..***, then the receiver will receive as many transfers as agent playing the sender role in the interaction. The receiver will wait for all agents to send the expected information. This implies all agents have to commit with the informatino deliver. When sender and receiver have cardinality **1..***, then each sender sends the information to each one of the receivers, and each receiver receives information from all senders. So, if there are 3 agents playing the sender role and 4 agents playing the receiver role, there will be 12 messages sent in total.

Each UIColaborates and UIInitiates permit to define futher conditions for information transfer. When these conditions are met in the sender side, the information will be delivered. When these conditions are met in the receiver side, the mental state of the receiver will be updated. Naturally, the conditions of the sender and receiver side can be totally different, but always referring to information already known by each agent.

By clicking twice on UIColaborates or UIInitiates relationships, a dialog like the one shown in figure 9 will appear. The dialog requests defining an entity encapsulating the conditions. The code generator recognises the GRASIAMentalStatePattern, which identifies the condition with an agent diagram.

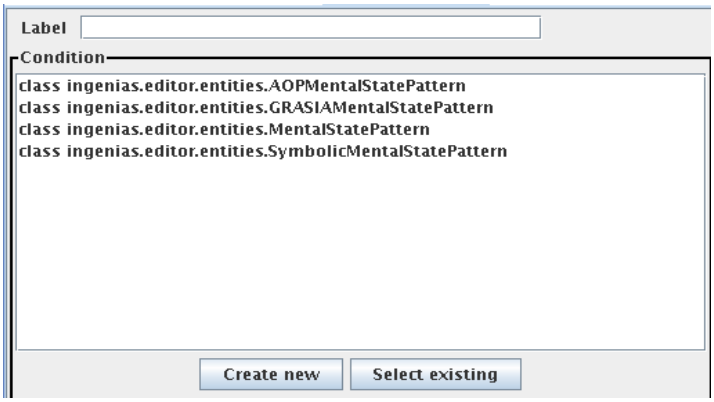

Figure 9: dialog for defining extra conditions in a UIColaborates or UIInitates relationship.

The condition should be expressed following the example from figure 10. The diagram must contain a ConditionalMentalState unit. This entity is associated to other mental entities if required. Each association can be marked with a label. This helps referencing later on this entity.

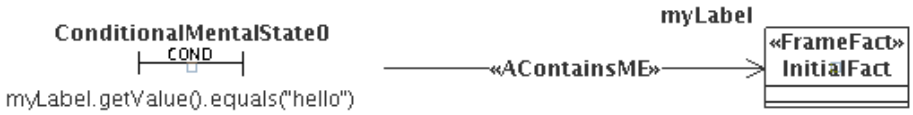

Figure 10: Mental condition for sending the information

These labels are defined following the set of submenus shown in figure 11.

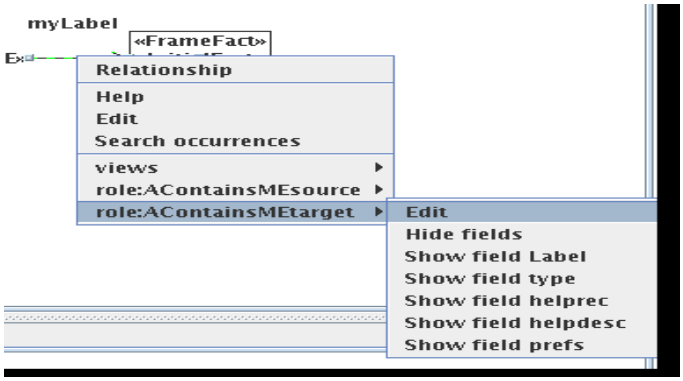

Figure 11: Assigning a label to a relationship end

Within the mental state condition entity, the developer is expected to define a JAVA valid boolean expression. Inside these boolean expressions, there could be references to the mental state entities associated to the mental state condition entity. Such entities will be accesible through variables named exactly like the defined labels. A developer should expect these variables to contain instances of mental entities exactly of the same type as those appearing in the diagram.

7.3 Storing information transferred

Each conversation is represented in the mental state with a *runtime conversation* entity. This entity has an attribute storing information to be stored aside of the mental state. Hence, each conversation acts as a repository of information.

When an information transfer happens, the requested information will be taken from this independent repository, or, if not found, from the general mental state. When the information is finally transferred to the receiver, it is stored in the mental representation the receiver agent has of the converation.

To the information stored within a conversation, only tasks associated directly in the protocol or linked with a *IAccesses* can obtain or modify information stored within a conversation.

7.4 Launching an interaction

The interaction is launched by means of a task and a conversation entity. The conversation represents an instance of an interaction in run-time. A conversation has, internally, a reference to the interaction class it represents and contains runtime information, such as the state of the interaction or the participanting members. It is important that the task is associated with a GTCreates relationship to the conversation, otherwise, the interaction will not be launched.

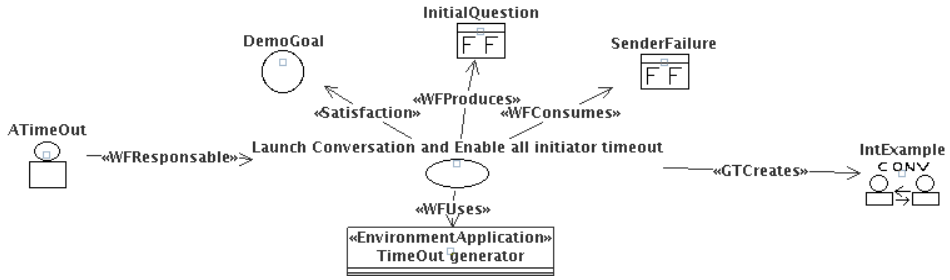

Figure 12: A task launching a conversation

The members of the interaction can be set within the task or, alternatively, use the default setup. By defaul, if the task does not modify anything, the JADE yellow pages service is used to detect agents playing the required roles. If many are available and cardinality of the corresponding role is not 1 in the interaction definition, then all of them are chosen. If the cardinality is 1, the first one is chosen.

The interaction launching task is owned by the initiator of the interaction. If the initiator wants to set up who will colaborate, then it is done, first, by associating a code component to the task in a component diagram.

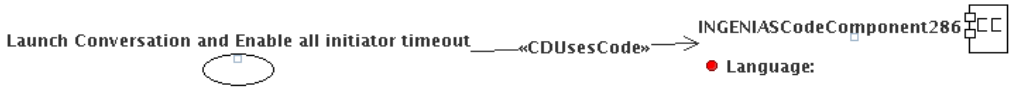

Figure 13: Associating code for selecting colaborators in an interaction

Within the code component, one can find code like this

```
outputsdefaultConversation.addCollaborators("C_2");

outputsdefaultConversation.addCollaborators("C_3");

outputsdefaultConversation.addCollaborators("B_1");
```

The addCollaborators expects a local id of the agent to colaborate with. As a result, when the task finishes, the agents whose ids are indicated within the task will be located. If those agents do not play all the roles required by the interaction, an exception will be triggered afterwards the task finishes. It is allowed that an agent plays more than one role in the interaction.

Here, it is assumed the ids of the agents are known, but they may not. That is the reason for having a reference to a yellow pages directory within the task generated by default.

7.5 States of an interaction

During the interaction execution, the state of the interaction can be consulted. Valid states are:

- FINISHED. The interaction has finished succesfully.
- RUNNING. The interaction is being executed.
- ABORTED. The interaction has been aborted due to one of the following reasons:
  - The information to be sent or received was not available within the defined timeout. A timeout of 0 means wait forever. Otherwise, time is measured in milliseconds. If the timeout occurs, the conversation state is set to ABORTED and the abort code to TIMEOUT. This situation can happen at any moment of the interaction.
  - The agents playing the required roles were not available. The conversation state is set to ABORTED and the abort code to NO_AGENTS.
  - There is another kind of error. The conversation state is set to ABORTED and the abort code to INTERNAL_FAILURE.

In any of the cases above, the conversation object can return the abort code with its method getAbortCode. Valid abort codes are defined into ingenias.jade.comm.Conversation entity.

- INITIATED. The interaction instance has been created but only locally. It is still to be known by collaborators.
- RUNNING. The interaction is being executed at this moment. If the agent is a colaborator, the conversation is known by this colaborator and the initiator, at least. If the agent is an initiator, the conversation can be assumed to be known by all colaborators.

The GUI debugging facilities of the IAF showd more intermediate states, but those are not available for the developer. These need not to be known by developers because they are handled automatically by the IAF framework. The concern of the developer is required only in the situations above depicted.

A task, in principle, does not have access to the information stored within a conversation. This is possible only when the task is associated to some of UIInitiates or UIColaborates relationships or when the task is linked with a IAccesses to the interaction entity whose running instance is being considered. Having access to this information, a task may decide to launch another interaction instance. In this case, there is a migration of the data contained in the initial conversation into the new one. This facilitates nesting protocols.

7.6 Modifying conversations

A task can be associated to different conversations of the same type. Figure 14 shows an example of this situation. A task magement gains access to all existing conversations of an interaction *IntExample*, whatever their state is, in the mental state of the agent. Once accessed, a task can insert or delete information from the local repository of the interaction. This can be used to interconnect to separated instances of the same interaction or to conduct the evolution of other conversations. A scenario where the later may be necessary involves an agent handingl several conversations of the same type. After exhausting the resources of the agent, a management task intervenes and cancels all prescindible conversations. This is done by inserting an AbortConversation fact, which according to the corresponding protocol of IntExample, makes the conversation progress to an end.

Though a task may abort a conversation, the mechanisms to implement this properly are not ready yet. Therefore, the only means to abort explicitly a conversation limit to those presented in section 7.5.

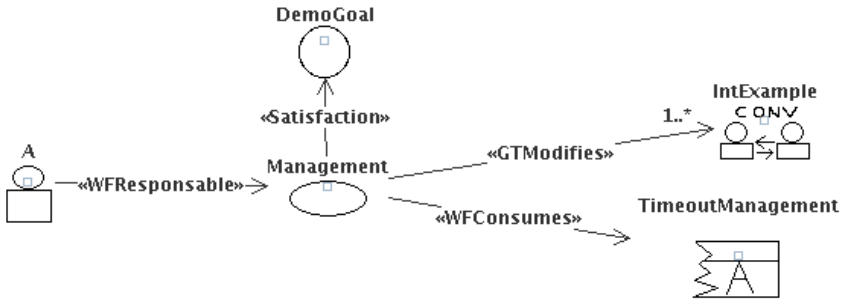

Figure 14: Task accesing existing conversations

7.7 Defining timeouts

In each protocol definition, a developer can define the timeout. The timeout can be defined in the sender side or in the receiver side of the interaction unit. It is defined selecting one of the sides and choosing the uiinitatestarget or the uicolaboratestarget role editing actions. The number it is expecting will be interpreted as milliseconds. If nothing is declared, an infinite timeout is assumed.

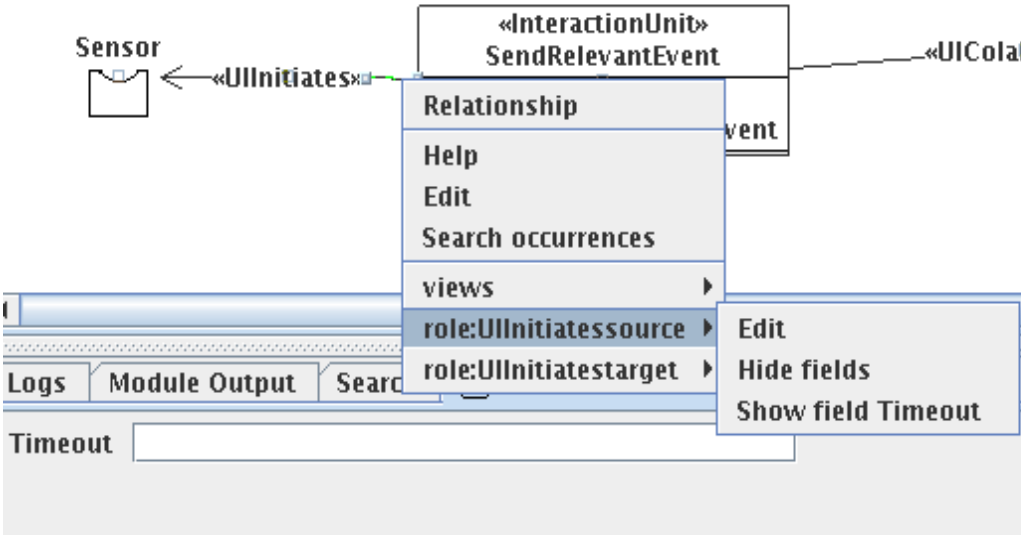

Figure 15: Timeout definition

7.8 When an conversation is aborted

An important part of the development consists in studying what to do with aborted conversations. Here, there are several policies:

- Delete the conversation and additional entities created during the process.
- Try again.

Since a task can have access to all conversations of a certain type, it is possible to react on aborted conversations. This task could collect all conversations of certain type and decide what to do .

8 Goals

Goals are interpreted by the IAF as *service goals*. A *service goal* is never destroyed/satisfied and they guide the task execution as follows:

- Select all tasks that can satisfy the goal
- Execute all tasks provided that required inputs exist

Those steps are executed continuosly. Future versions of the IAF will consider other kinds of goals.

9 Tasks

A task is the abstraction of a set of actions performed over the environment and/or the agent mental state. The modification in the environment happens with the intervention of applications. Modification of the mental state is controlled in the specification, as it will be explained in this section.

An agent starts tasks because two conditions concur:

- There exists instances of the information required by the task
- The agent is pursuing a goal that the goal can attain

Tasks are initially scheduled for execution in the deliberation stage. They are executed one by one, therefore it is not necessary to worry about concurrence. Nevertheless, there is a concern about the arbitrary execution order of tasks, which depends on the moment they were scheduled. Also, before execution, inputs are checked again, just in case any previously executed task modified them.

After execution, the task will have produced evidences, modified, or deleted existing mental entities.

9.1 Inputs

A task obtains necessary inputs from two places:

- From the mental state exclusively. These are the non-conversational tasks. The task obtains inputs by reading the mental state.
- From a conversation, first, and from a mental state, second. Each conversation (an interaction in runtime) has a conversation workspace where information received from other agents is stored (read section 7.3 for more details). Conversational task are said to be binded to conversations. Binding happens in two ways:
    - the task is associated to an interaction unit and a role/agent by means of a ternary relationship (*UIInitates* or *UIColaborates*) in a protocol definition.
    - The task is associated to a conversation with a *IAccesses* relationship

A task not binded in any way to an interaction cannot access the information stored within conversations. Once clarified the places the task can take information from, it is time to consider what kind of inputs can be defined. A task can have the following inputs:

- Framefact. It represents a piece of information, probably further detailed with slots.
- Goal. A goal of the agent.
- Event. An event produced by an application.
- Conversation. An interaction during its execution.

These inputs can be associated to the task by the following relationships:

- WFConsumes. The task take the input from a conversation to which the task is binded. If the information is not within the task, it is taken from the mental state. As a result, the information is deleted from its original location when the task finishes.
- Consumes. The task takes the information from the mental state only. When the task finishes, the information is deleted.

- GTModifies. The information is read from the mental state or from the conversation to which the task is binded. The information is not deleted, but can me modified by the task.

In the definition of a task, it can have inputs with a cardinality label, either **1**, **1..*,** or **0..***. Cardinality **1**, means is is needed at least one instance. Cardinality **1..*** stands for collecting all instances, requiring at least **1** instance to exist. Cardinality 0**..*** stands for collecting all instances, not requiring any instance to exist. Cardinality is defined as in figure 11.

9.2 Outputs

The task generates as output those entities connected with the following relationships:

- *WFProduces*. An instance of the entity is produced and stored in the conversation to which the task is binded, if there is any. If there is none, the information is stored in the mental state directly. If the task produces new conversations, the information is stored in those conversations as well. Conversations cannot be created with this kind of association.

- *GTCreates*. An instance of the entity is produced and stored in the mental state. Conversation entities can be created only with this kind of association.

Unlike inputs, outputs have no cardinality. A task can produce as output Conversations or framefacts. At this moment, the task cannot produce goals, and event production is limited to applications.

9.3 Generating a task code

To properly generate a task, it is needed:

- To link the task to a role or an agent.

- To link the task with a goal by means of a *GTSatisfies* relationship

- To have at least one input. An input is a mental entity connected by means of *WFConsumes, Consumes,* or GTModifies.

- Optionally, to have some output. An output is a mental entity connected by means of *WFProduces* or *GTCreates*.

- Optionally, the task can be associated to Applications. This indicates that the task requires to have access to an instance of the application. In terms of code, the generator produces automatically a reference to the application so that the programmer can access to its API.

9.4 Referring to other conversations

It is possible to refer to other conversations from a task. For this, it is sufficient to connect with GTModifies the task with the different conversations to be obtained. If cardinality attributes are used, all the conversations of a given type will be retrieved. If no cardinality attributes are set, then one of them will be chosen.

This is useful to take decisions about other running conversations, since a task can remove or add new information to them.

Adding information is possible with the method *addCurrentContent* of the conversation. Also, *removeCurrentContentElement* will permit to remove an entity if its id is known. Method *getCurrentContentElements* will return an Enumeration of the elements contained.

9.5 Tasks and memory management

The tasks produce entities that may flood the mental state of the agent with entities no further used. To prevent this, there is a special task in charge of deleting all information produced by tasks which are not used anymore. Also, produced conversations that are finished or aborted for more than 10 seconds, they are deleted automatically.

Nevertheless, the definition of a task should be made carefully. A dangerous situation happens when the task has only GTModifies relationships connecting the task to different entities. This imply the task will be scheduled again and again as long as those entities exist. This may be a desired behavior from the developer, nevertheless, it is strongly advise to use at least one of the following relationships Consumes or WFConsumes, so that task scheduling happens only once. These relationships imply removing the entity, so next activation of the task requires regenerating the removed entity.

9.6 Self awareness

The task, sometimes, requires to know who the agent is. For instance, a task generates a log of its activities. The logging system requires an agent id to properly associate this task's outputs to the corresponding agent.

For these cases, the developer can create a framefact entity called exactly **Agent_data**. The code generator recognises this name and will not proceed as with the other entities. An **Agent_data** contains the id of the current agent. This entity must not be consumed, since it may be used by other tasks. Instead, this entity should be connected by means of a **gtmodifies** connection.

9.7 Generated code of a task

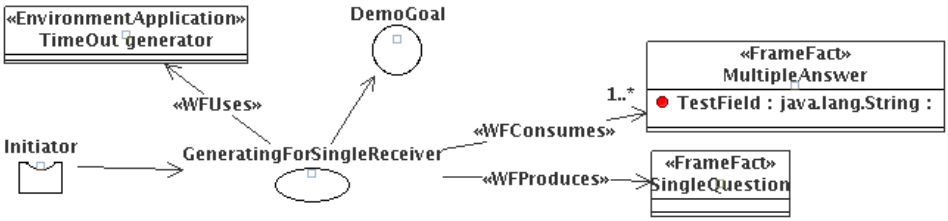

Figure 16: Example of task specification

According to previous indications, the code produced for a task like the one shown in figure 16, would be something like this:

```
package ingenias.jade.components;

import java.util.*;
import ingenias.jade.exception.*;
import ingenias.jade.comm.*;
import ingenias.jade.mental.*;
import ingenias.editor.entities.*;

public class GeneratingForSingleReceiverTask extends Task{

       public GeneratingForSingleReceiverTask(String id){
```

```
                super(id,"GeneratingForSingleReceiver");
        }

        public void execute() throws TaskException{
                Collection<MultipleAnswer>  eiMultipleAnswer=new
Vector(this.getAllInputsOfType("MultipleAnswer"));
                TimeOut_generatorApp
eaTimeOut_generator=(TimeOut_generatorApp)this.getApplication("TimeOut_generator");
                // This means that the task participates in the interaction IntExample
                RuntimeConversation
conversationContextIntExample=this.getConversationContext();
                Vector<TaskOutput> outputs = this.getOutputs();
                TaskOutput defaultOutput= outputs.firstElement();
                TaskOutput      outputsdefault=findOutputAlternative("default",
                                outputs);
                SingleQuestion outputsdefaultSingleQuestion=
                        (SingleQuestion)
                        outputsdefault.getEntityByType("SingleQuestion");
                YellowPages yp=null; // only available for initiators of interactions


                // HERE YOUR CODE GOES

        }

}
```

Since the MultipleAnswer fact had multiple cardinality, the task code defines the variable eiMultipleAnswer and stored into it a collection of the existing instances of MultipleAnswers at the time the task was scheduled. These entities can be consulted with a common iterator.

Also, the task has access to an application of type TimeOut_Generator. Note that the blank space has been replaced with an underscore during generation. The reference to the application is stored in the eaTimeOut_generator variable. Methods defined in the class TimeOut_generatorApp (see section 10) are available to the task.

This task in concrete is binded to an interaction since it is used in a protocol (see figure 7). This permits to the task to access other relevant information stored in the conversation and modify it, though it is not recommended to do so with a binded conversation, since it violates the declaration of the task. One of the strengths of the task declaration is the explicit definition of its inputs and outputs, determining this way its scope. Nevertheless, the conversation stores other information which makes sense to know, like, the role this agent is playing in the interaction, or which are the other participants.

The YelloPages variable is not enabled here since its availability is limited to tasks starting interactions. A yellow pages object permit to consult the yellow pages directory of JADE to locate other agents. This is necessary when the task needs to determine who will participate in the interaction. Please, consult section 7.4 for more information.

Outputs of a task are codified inside defaultOutput variable. In the future releases, a task can produce different sets of outputs. Right now, the developer has to use the defaultOutput set. The code generator generates references to the objects that will be produced when the task finishes. Object creation and their insertion into the mental state is automatic. The task has access to them for two reasons:

- To properly initialize them. By using set/get methods (see section 6.2) for more information.

- To decide if they are produced, after all. The task can abort the object production by removing the corresponding object.

An example of the later is the following code

```
outputsdefault.removeEntity(outputsdefaultSingleQuestion);
```

This code will remove the object referred by variable *outputsdefaultSingleQuestion*, i.e., the task will not produce the *SingleQuestion* instance the system is waiting.

Removing an output is reasonable when the task produces several types of entities depending on the circumstances. For instance, in a request for proposals, if the proposals are not satisfying, the task may produce another conversation to gather more proposals; but if they satisfy some developer's criteria, then the conversation launch is cancelled by removing its corresponding output.

Future releases of the IAF will provide a more convenient solution to this problem.

9.8 Allocating the modified code into the specification

During implementation, it is common to generate several times the code corresponding to the specification. This may delete the modifications made in the task. To prevent this, a mechanisms for storing modified task code has been devised. It requires a component diagram and associating the task with an INGENIAS Code Component entity, just like figure 17.

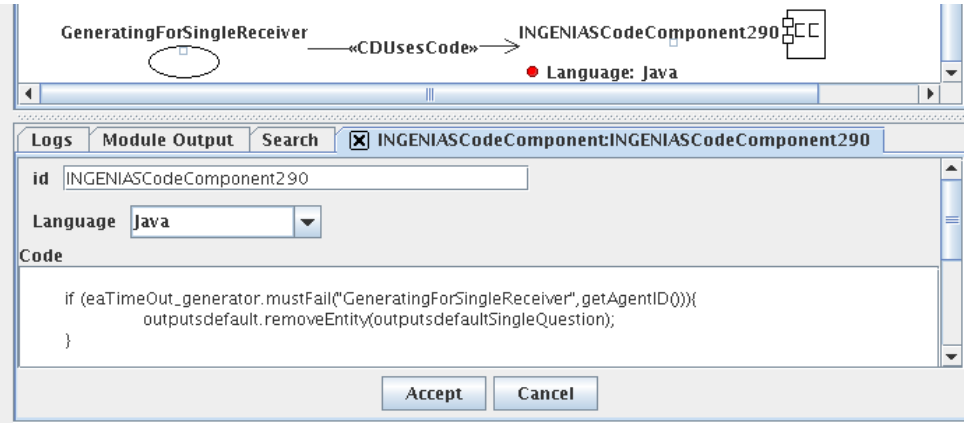

Figure 17: Code associated to a task

By clicking twice on the INGENIAS Component, a developer should copy and paste the code inserted in the task. This code corresponds to the one replacing the comment ***here your code goes*** of the previous example. If it is inserted, then, the next time the IAF generates code, this code will be inserted in the proper place. Other modified code, like new methods, changes in the class names, new attributes, or new imports, will be ignored. Therefore, a developer should use always fully qualified objects in the code stored in an INGENIAS Code component.

To conclude, a task can be associated only to an INGENIAS Code compoment.

10 Integrating with external components

The integration of non-agent software into the INGENIAS Agent Framework is made by means of Applications. Applications can be external or internal. External means that they exist already prior to the creation of the system. Internal means they have to be tailored ad-hoc for the current development.

In any case, an application may be understood as a wrapper to include non-agent software into the specification. The declaration of an application starts by associating an application to the agent, as it is done in Figure 18.

Figure 18. Application to agent association (environment diagram)

Applications are linked to a file representing its java interface. This is done in a component diagram by associating an *INGENIASComponent* to the application. The diagram to be obtained is shown in Figure 19. The creation of this diagram requires creating an INGENIAS Component and associating it to the application.

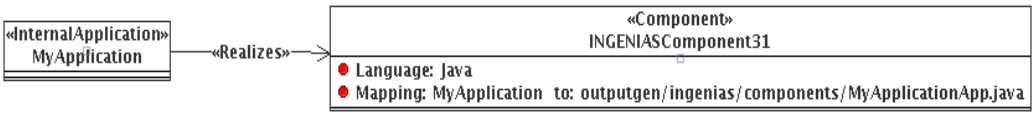

Figure 19.The description of a component realizing an application

Then, inside the INGENIAS component, a *FileSpecPatternMapping* element must be created in the files field. This new element must point at the recently created *Application* as Figure 20 shows.

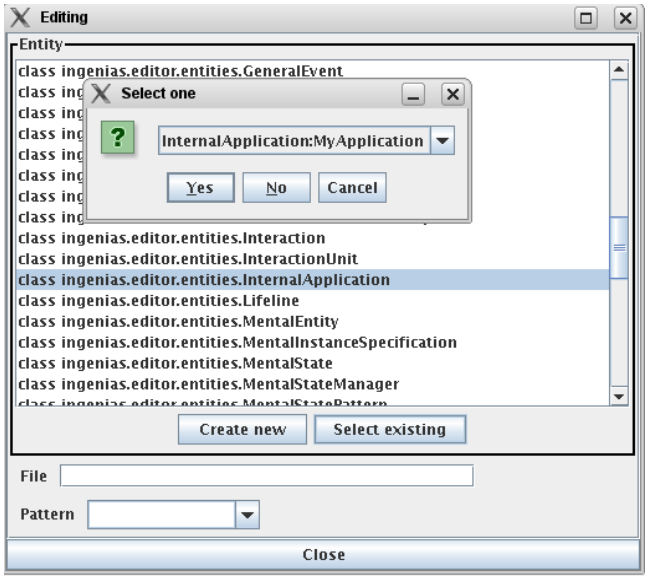

Figure 20.Selecting an entity to be represented by this FileSpecPatternMapping

Once selected the application, it is time to type down the number of the physical file that this component is using and the followed pattern in the implementation. There are two possible patterns, the singleton pattern and the multiple instances pattern. A singleton pattern means there is only one instance of the application in the system and anybody having access to it, i.e. having one as indicated in Figure 18, will access the

same instance. The other pattern is the opposite, i.e., anybody having access to it will access another different instance.

Depending on the needs, a singleton or multiple instances approach can be used:

- If the application needs to be accessed externally by other agents or classes, use a singleton pattern

- If the application handles delicate data that should be kept private for the sake of encapsulation, use a multiple instances.

The file field should point at the file containing the interface declaration of the application. This corresponds to the form *outputgen/ ingenias/components/_**APPLICATION_NAME_HERE**_App.java*. This part is important only if the ApplicationLinker plugin is going to be used. Otherwise, what this field contains does not matter much, expect for having a better specified system.

By default, if the File field is empty, the code generator fills in this field with the default output location.

An application requires three files in the IAF code generation and they are all allocated in the same folder:

- ***_APPLICATION_NAME_HERE_****App.java*. It contains the interface the application offers to the system. It is **not overwritten** with a new code generation.

- ***_APPLICATION_NAME_HERE_****AppImp.java*. It is the implementation of the interface.It is **not overwritten** with a new code generation.

- ***_APPLICATION_NAME_HERE_****AppInit.java*. It is the initialization code of the application. It is invoked only once if the pattern is singleton or several times, in the case of a multiple-instances pattern. It is **overwritten** with a new code generation.

10.1 Initializing and terminating Applications

This initialization/termination is necessary in some kind of applications, like those representing GUIs or information sources. Initialization and destruction code is defined in a specification with a component diagram.

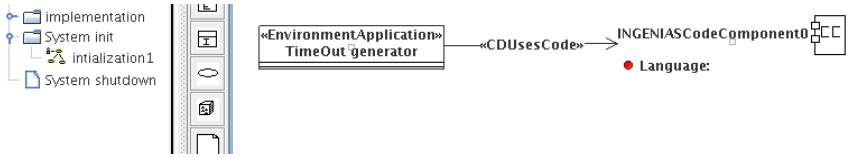

Figure 21: Definition of initialization code

To initialize an *application*, it is required to define a CDUsesCode relationship in a component diagram between an INGENIAS Code component and the application.

This code is invoked in a different way depending on the kind of pattern used for the application:

- A multiple instance pattern. Then, each time a new application is created, the code is invoked. Application are created once for each agent.

- A singleton pattern. The code is invoked the first time an agent gains access to the application.

10.2 Code within the applications

Typically, one wants to produce information from an application and inform the agent. This is modelled with the perception relationships.

To create and assert new information in the owner of this application, the following can be used.

```
SensorInformationUpdated nevent= new SensorInformationUpdated();

try {
      this.getOwner().getMSM().addMentalEntity(nevent);
} catch (InvalidEntity e) {
      e.printStackTrace();
}
```

This works well when there is actually a single owner. An application producing events may be associated with different agents as a singleton pattern. This is not recommended since it induces failures detecting who the owner is. **A singleton application used by several agents should produce no events.**

It is important to use the empty constructor of the new mental entity, since this constructor chooses a non-conflicting ID always.

11 Agent perception

Agent perception is defined by means of association with applications. Perception happens by inserting a piece of information into the agent mental state. Usually, the piece inserted is an event type, concretely, an ApplicationEventSlots, which allows to define different attributes of the event to be defined.

11.1 Defining the perception relationship

For a perception it is required the agent has one association *application belongs to* with the application and an *eperceivesnotification* relationship.

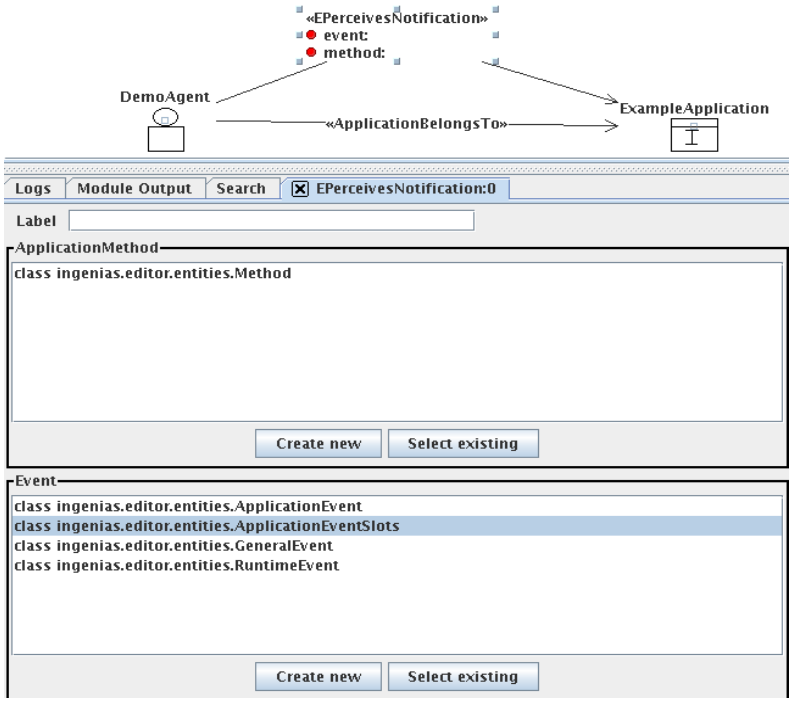

Figure 22: Defining a perception relationship

The developer is expected to either create a new event type or select an existing one. The IAF will recognise the ApplicationEventSlots type, so choose this one.

For demonstration purposes, the example defines an ExampleEvent of type ApplicationEventSlots (see figure 23).

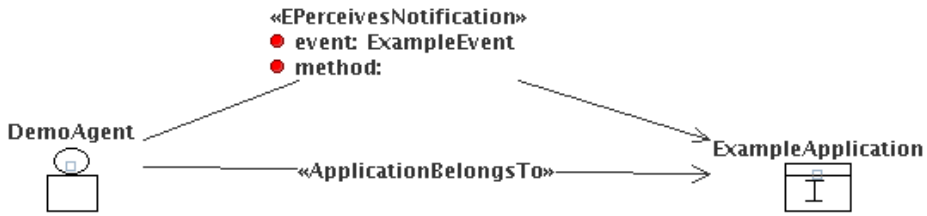

Figure 23: An event type as been associated to the perception

The developer can define as many eperceivesnotification relationships as needed (see figure 24).

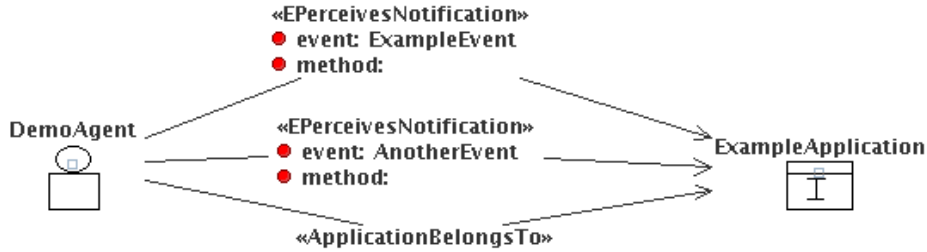

Figure 24: Another event is defined

11.2 Implementing perception

The part to be implemented by the developer concerns only to the creation of the event and its insertion in the mental state.

This has been already addressed in the section 10.2.

11.3 Perception in runtime

During runtime, the perception relationships are automatically realised by the IAF GUI.

Figure 25: Perception relationship between an agent and a GUI

The IAF generates dummy compoments that are able to produce the evidences pointed out by the relationships as shows figure 26. For more details about the purpose of this view, consult section 14.

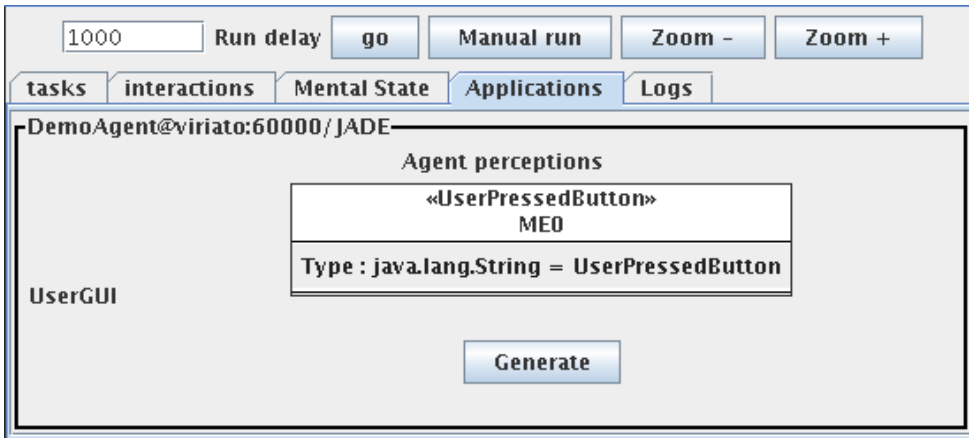

Figure 26: Application view of the generated system

12 Creating deployments

The creation of specific deployment configurations is made through the *DeploymentPackage* entity. This entity is later transformed into a set of scripts that launch the indicated set of agents.

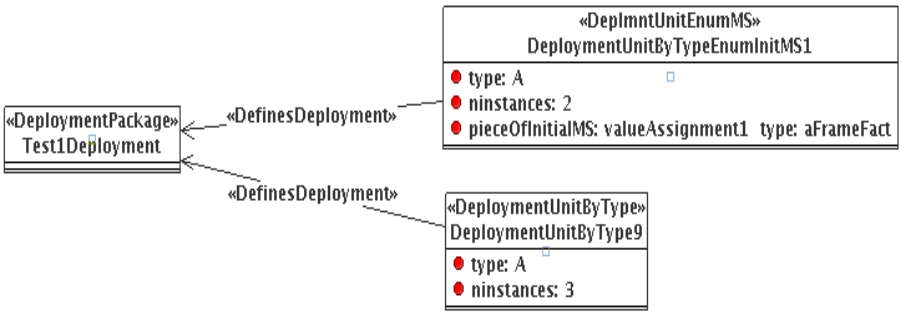

Figure 27.A deployment configuration example (deployment diagram)

The diagram definition of a deployment permits easily to altere it by removing/adding edges to existing deployment units.

There are two ways of configuring a deployment. The first uses a *DeploymentUnitByType*. This entity indicates the type of agent to exist in the running system and the number of instances launched from that type. Once launched, the system should have 3 instances of agent A with an initial mental state extracted from any agent diagram showing a direct association between agent A and a *Mental State* entity.

A deployment package entity can define extra parameters for further customization. The IAF recognises two names of parameters, namely **port** and **memory**. The first will be used to change the default port where the JADE platform is, and the port to wich the JADE container will connect. This is useful when it is planned to launch several MAS in the same machine or when the port is already occupied by another application. By default, **port** has the value of **60000**. The other parameter permits the developer to define how much memory the MAS is allowed to use. By default, the **memory** has a value of **128m**. The **m** stands for megabytes.

About the initial mental state each agent will have, by default, the agent is created with empty instances, i.e. with slots not initialised, of the mental entities associated to the agent by a MentalState entity instance.

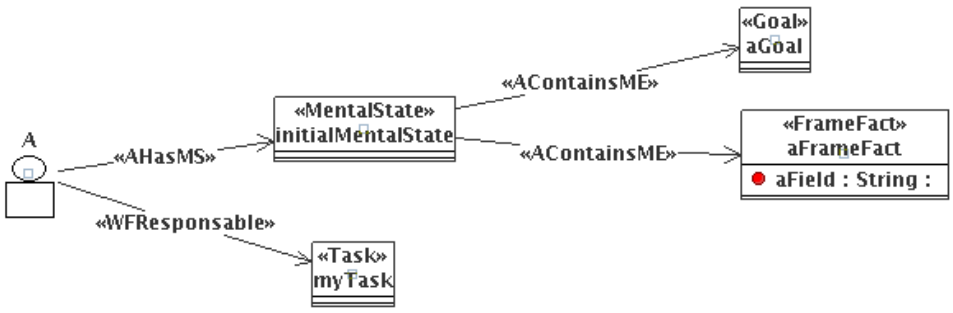

Figure 28.A deployment configuration example (agent diagram)

In the diagram, agent A is supposed to have an initial mental state made of one instance of aGoal and one instance of anotherFrameFact. Note that the instance of anotherFrameFact should have an attribute myAttribute with some value. In this case, the value will be null. If the developer wants to assign different values to attributes of instances, another deployment unit type has to be used, the the

DeploymentUnitByTypeEnumInit, which corresponds to the second way of configuring a deployment.

The *DeploymentUnitByTypeEnumInit* indicates not only how many instances of an agent should be created, but what initial mental entities it should have and what values they should have assigned, if they needed so. This is more precise than the previous approach, but also more tiresome.

Let us assume that the initial state of an agent of type ATimeOut requires incorporating a new FailAllParticipant2 instance and initialising the field aField of this instance to "hello".

First, a DeploymentUnitByTypeEnumInitMS instance is created. Clicking twice on it, the following properties window appear.

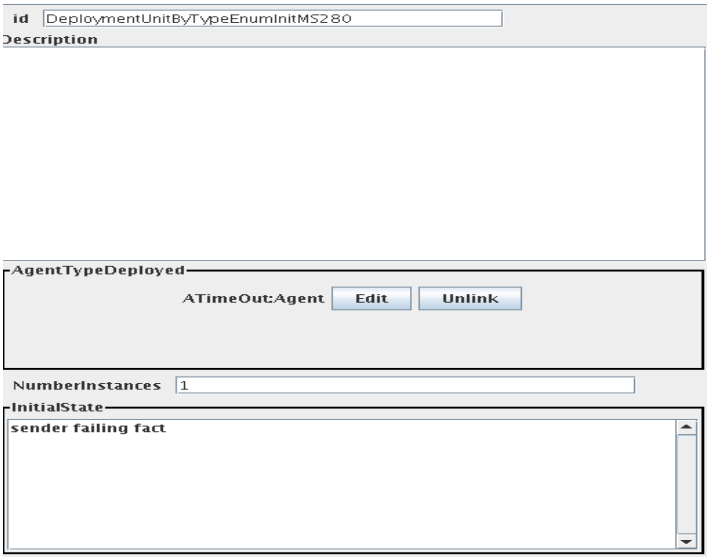

Figure 29: Creating an instance of a deployment unit by type with initial mental state specification

After choosing the agent type, ATimeOut, we proceed to add the new mental entities instances required in the *initial state* box. If one of these entities is created, the following window appears:

Figure 30: An instance of a mental entity is being defined within a deployment unit

Here, a slot value has to be defined, since the initial purpose is to set the aField slot to a "hello" value. To do so, a new slot value is created in the *slot value* box. As a result, the following window appears:

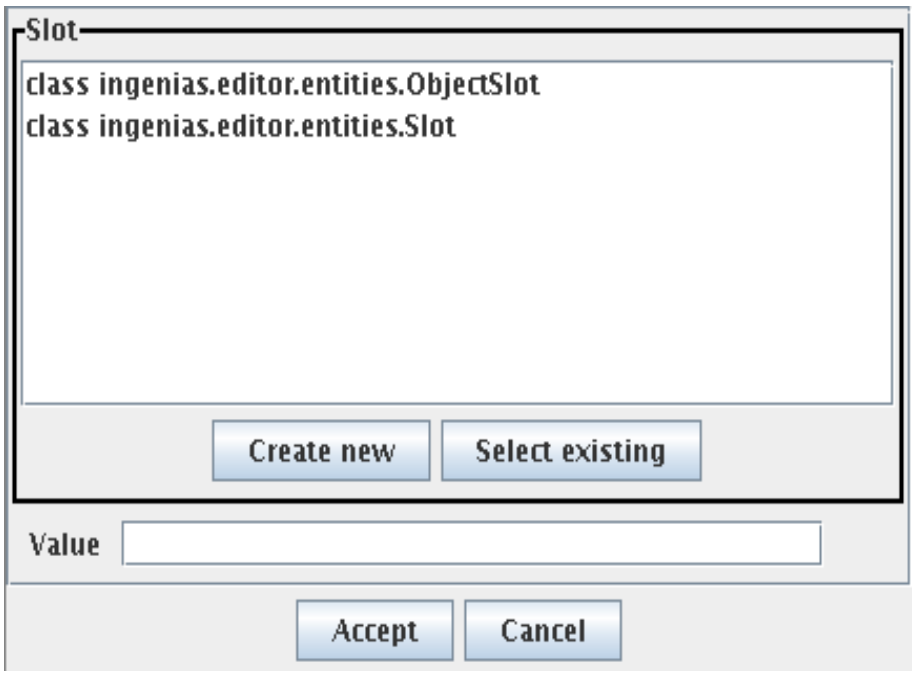

Figure 31: Selecting a slot of an instance for value assignment

An existing slot or a new one has to be selected. Whatever slot is chosen, it must be one associated to the entity to be instantiated. In this case, the *aField* is searched and found.

Figure 32: The field has been selected and a value has been typed down.

It remains only to type in the expected value and accept in each opened window. This value will be added to the instance using a setter method, like *object.setAField("hello")*. After this action, the new instance has been defined, though others could be defined as well.

The final result is the following:

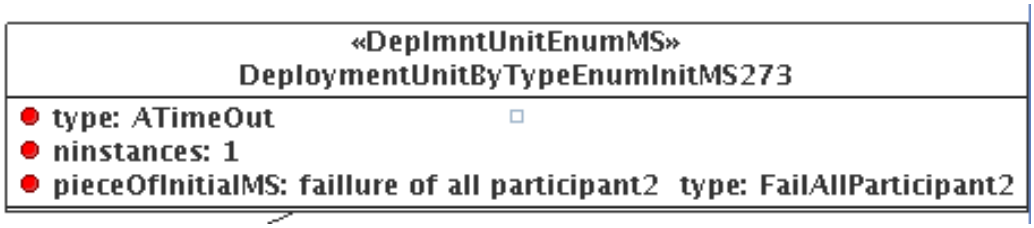

Figure 33: Deployment unit by type enum init completely defined

12.1 Deployment folder structure

When code is generated, each deployment configuration is associated to a target in the launch.xml file (you can find this file in the genoutput folder).
The default configuration of code generation produces two folders: **permsrc** and **gensrc**. The first contains files that are generated only once. They correspond to classes to be modified manually by the user or created manually. These classes are not overwritten by a new code generation. The folder **gensrc** contains all those classes generated automatically
There is as well a folder for the main project files, the src folder, common to most developments.

The different bundles that can exist within src are automatically copied to the binary folder.

12.2 Build files and special targets

For each generated MAS, there is a build.xml generated automatically. This build file will be used for launching the MAS in different ways.

- There is a target inside the build.xml for each to launch a MAS following a concrete deployment configuration. Each deployment confing may define its own memory and listening port parameters.
- For each deployment configuration, there is a special target ended in **ProdStandAlone**. This target allows to launch together in a single action the JADE platform and the MAS. This one is intended for production stages.
- If tests were defined, there are special targets to launch them as well as Junit tests one by one or all at the same time.

13 The default system

Once generated the code, a developer finds a semi-working system.

- The launch.xml contains the code for launching the MAS. There are as many options as deployment configs defined in the specification.

- Tasks produce automatically the desired output whenever the required input exists. Nevertheless, there is no internal process that transforms the inputs into the expected outputs. In fact, generated outputs are empty. For instance, let us supposed a task consumes an entity called aFrameFact of type FrameFact, it produces a FrameFact entity called anotherFrameFact, and this anotherFrameFact entity has a slot called mySlot. In the default running system, the task, whenever an instance of aFrameFact is found, it produces an instance of anotherFrameFact with a slot called mySlot containing nothing.

- The initial mental state is that determined by the deployment configuration.

- The applications defined in the specification do nothing. Events originating from those applications can be replicated by means of the default graphic interface, though. The applications code will have those methods declared in the specification.

13.1 Providing the final code

A developer is supposed to provide the following information

- In a task, determine the code that transfers information from the inputs to the outputs. This code is inserted in the specification in form of INGENIAS code components. This way, when the code of the task is generated again, it will insert this code in the proper place.

- In an application, determine the code that corresponds to the individual methods defined in the specification. Perhaps, during the development, new methods are needed. In this case, and if java language is used, the application linker can help to keep the specification updated. Also, define the code for initializing/terminating the application.

- Additional classes. Other classes needed by the developer can be created anywhere, just as in any java development. As an advise, we recommend creating them in a separated folder, like the classic src. This way, code generated by the system will be completely separated from other code.

14 Debugging

Files and logs. Search through the logs. Replacement of blanks with underscores (if not taken into account, searchs may fail)

14.1 Windows of the debugger

By launching a deployment, the developer starts by default a graphical environment that permits to inspect the MAS behavior.

The interactions view shows the states of the protocol associated to the interaction. The initial protocol definition is transformed into a set of independent state machines whose execution is what the developer sees. These state machines can be in multiple states at the same time. Current states are shown in red. When the rightmost label appears in red, it is interpreted as reaching the end state.

In the border line of the protocol, there is a label indicating the conversation ID and the name of the interaction as it appears in the specification. Inside the border, there are different boxes showing the individual state machines. Over each state machine, there is a label to identify the owner.

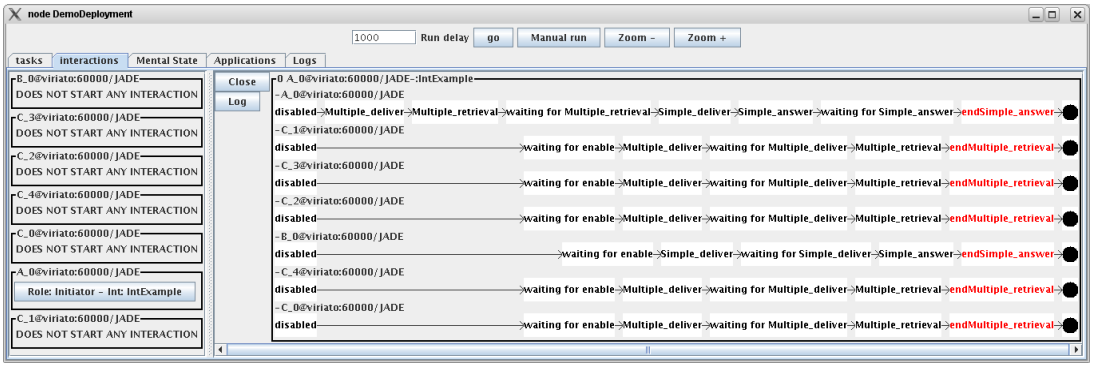

Figure 34: Running interactions

The logs window gives access to a human readable history describing all actions commited so far. The window has two parts. The list at the top contains the identifiers of the observed events. All events are first labelled with the name of the agent which generated them. After this name, different continuations may follow:

- **MSP-TaskIdentifier**:TaskType. This comes from a task execution.
- **MSM**: It refers to a modification of the mental state
- **An identifier of a conversation**. It informs of events associated to the conversation, like transfer of information or its current state.

If the developer chooses one of the elements of the list, the panel at the bottom will show content. Each entry has a limited size of 100 events. When the limit is reached, the oldest entries are deleted to leave space for the new ones.

When the developer just wants to find through the whole log, the filter field has to be used. Any text typed into this field will invoke a search through the whole log looking for the occurrence of the text in the id of the event or in the event content. Matching entries will be displayed to the user.

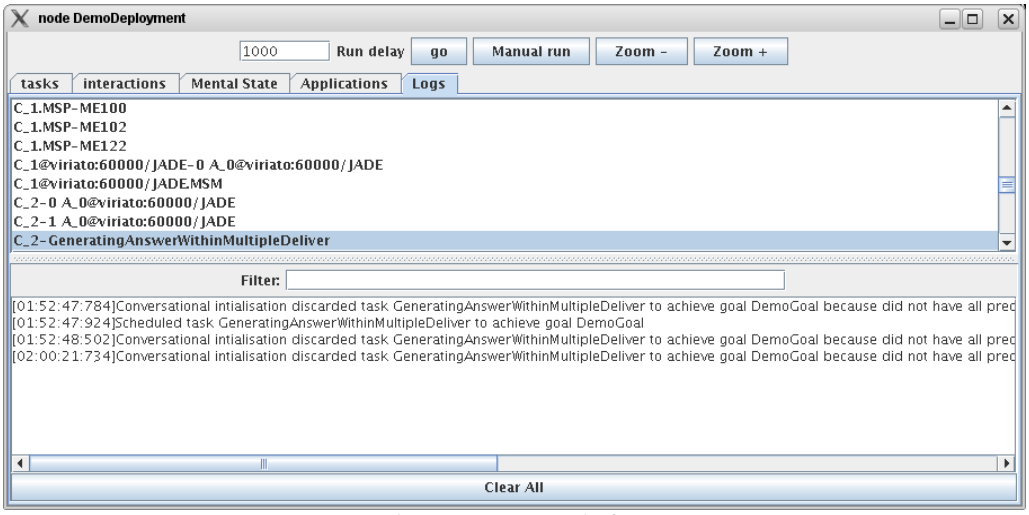

Figure 35: Logs window

The application view permits to simulate the generation of events to be perceived from the application. For each perception relationship, a new button will be shown. When the button is pressed, the event will be inserted in the agent owning the application. The name of the owner appears in the line border surrounding the application. This will work only to applications using the multiple instances pattern.

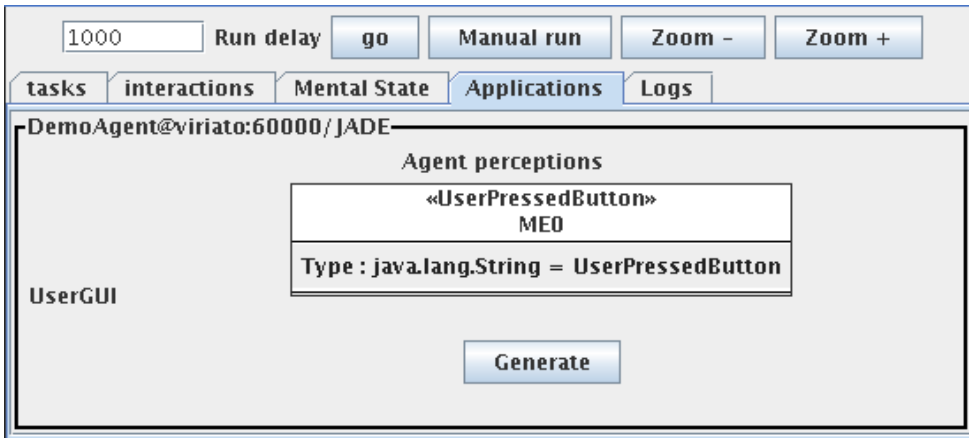

Figure 36: Application view

The mental state of each agent can be monitored from the mental state view. The mental state is shown when the user presses the corresponding button.

Figure 37: Mental State view

The tasks view permits to analyze the behavior of the agent step by step. For more details, read section 14.2.

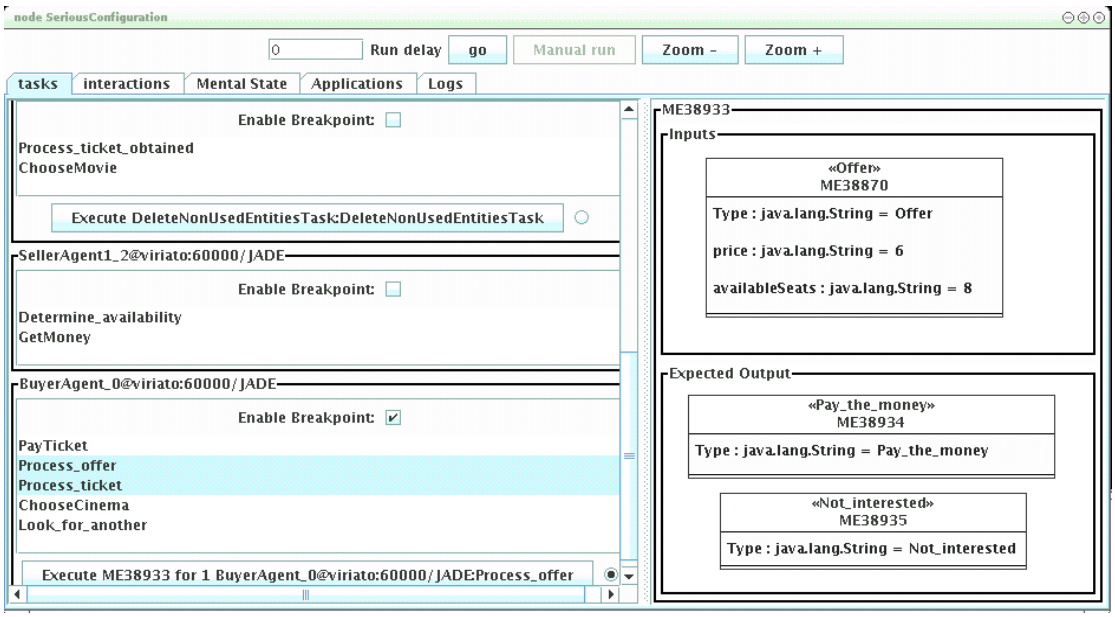

Figure 38: Tasks view

14.2 Step by step

First, the user must press the manual run button. This will prevent the mental state processor of the agent from making decisions. Instead, the user will be responsible of choosing what task to execute next.

Each agent is assigned a panel where tasks to be executed appear. As tasks are eligible for execution, they will be added to the corresponding agent box.

By choosing the radio button on the right side of the button, the inputs and outputs of the task are shown.

The idea consists in inspecting the agent mental state while no further actions are executed. If the mental state is known, it is easy to decide if the current scheduled tasks match the developer's expectations.

When the user finishes the inspection, there are several courses of action:

- Clicking on the task button. This will force the task execution, consuming or just accessing the input, and producing the output declared with the task.

- Clicking the **go** button. This will execute all scheduled tasks and will resume the mental state processor activities.

Taking to the desired task is not trivial in most cases. To help in this task, an equivalent of a breakpoint at the level of tasks has been devised. For each agent, a list shows which tasks are known. By choosing one and enabling the breakpoint checkbox, the execution will stop when the time of executing the selected task comes.

14.3 Logs

Besides the information accesible from the IAF GUI, which is limited, all generated logs are saved as well in files accesible anytime. These files can be found in the **log** folder which will be created after a first run.

Log files are named with the prefix **log** plus a number representing the time in milliseconds when the log file was created. Higher numbers mean newer logs.

Log files are unbounded and may grow until their size reaches the Giga scale. Hence, it is advisable only for development stages and not for production.

Logs are enabled with the ingenias.jade.LogFileOn set to true. Read section 15 for more information.

14.4 Mental Entities Traces

Each fact remembers what actions where applied over it. Concretely, the creation and transfer operations. This information is useful to trace back the information to its origin.

To acces this information a developer must include the following call in the code of a task. Tasks are the only place where this service is applicable.

If a developer included this code in a task containing an adequate fact type, like this one:

```
public class AnotherTaskTask extends Task{

      public AnotherTaskTask(String id){
            super(id,"AnotherTask");
      }

      public void execute() throws TaskException{

            DemoFact
eiDemoFact=(DemoFact)this.getFirstInputOfType("DemoFact");
            Vector<TaskOutput> outputs = this.getOutputs();
            TaskOutput defaultOutput= outputs.firstElement();

            TaskOutput  outputsdefault=findOutputAlternative("default",
                        outputs);

            YellowPages yp=null; // only available for initiators of
interactions
            DebugUtils.printStackTrace(eiDemoFact);
      }
}
```

A trace that might be obtained could be this one:

```
      [java] Stack print for fact ME11:DemoFact
      [java] 17:28:17 Creation by DemoAgent in ME10:HelloWorldTask
```

14.5 Mental Entities timestaps

Given a collection of instances of an entity, a developer may need to distinguish which entity existed before. To obtain this information, the stack of operations of the entity can be consulted. Each operation has a timestap in milliseconds that can be consulted.

15 Setup

The system can be configured by modifying the **config/Properties.prop** file. This file contains two lines:

```
ingenias.jade.LogFileOn=false;
ingenias.jade.GUIOn=true;
ingenias.jade.GarbageCollectionInterval=10;
```

These properties indicate, respectively, if logs have to be saved to files and if the GUI should be activated. Depending on a development or production environment, one or the other should be configured. Both have pros and cons:

- ingenias.jade.LogFileOn. If enabled, it stores all the information about the internal behavior in log files. Nevertheless, these files can grow until gathering all the remaining free hard disk. Due to the amount of information stored, and being an I/O operation, this slows down the execution.

- ingenias.jade.GUIOn. If enabled, it provides lots of useful information, besides permitting the step by step execution. Nevertheless, it slows down the MAS considerably, besides increasing the amount of required memory.

- ingenias.jade.GarbageCollectionInterval. It defines the amount of time between each mental state check in seconds.

16 Testing

The IAF code generation has a partial support for testing. It bases in the JUnit framework for testing. The testing is a blackbox one. Basically, it consists in determining, from an initial state, what the final state will be. The state here is made of the union of the individual mental states of every agent of the system.

Developers have to define their own tests, though the IAF provides some help in the following ways:

- Generation of skeleton test files. These files contains examples of asserts that can be used by the developer.

- Generation of initial setup of the sytem. This means initializing individual agents with their corresponding initial state. This is similar to the deployment definition.

- Generation of java classes for launching collection of tests. Usually, a developer wants to launch a sequence of chained tests to drive the system to determined conditions.

All tests are defined with the JUnit 4 testing framework. It is highly advised to read the fundamentals of JUnit testing before engaging into this section. Also, note there are strong differences between JUnit 4 and previous versions.

All tests presented here disable the graphics option by default. This is highly advisable for production and for testing, since it makes everything go faster. If the developer wants to reenable the visual inspection facilities, it is recommended to look at the files stored in the ingenias.testing package and set the following code:

```
IAFProperties.setGraphicsOn(false);
```

to

```
IAFProperties.setGraphicsOn(true);
```

16.1 Defining tests in the specification

The definition of a test starts by declaring the test itself in a component diagram

Figure 39: Declaring a test

The INGENIAS component point at a java class representing this test.

Then, the developer defines in a deployment diagram, what test units should be considered and what deployment configuration is used:

Figure 40: Declaring a collection of tests

The definition of a testing package requires choosing the deployment setup, to be defined in the same deployment diagram or another different one, and the collection of test units to apply. This permits reusing the same setup for different testing packages.

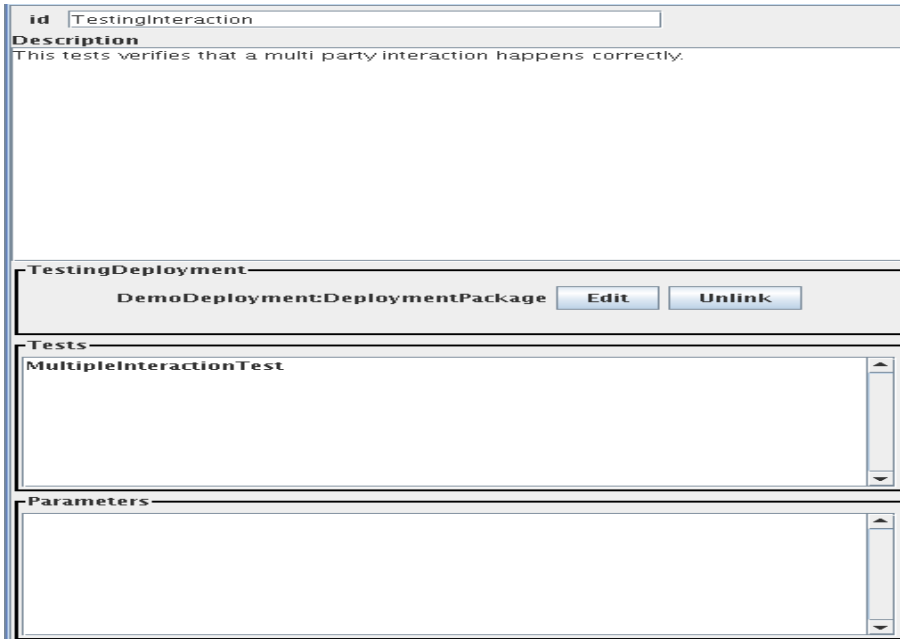

Figure 41: Definition of a collection of tests, properties view

If the user wants, more custom parameters can be added so that developers can customize the code generation.

16.2 Including code in the test

Inside a test, the following functionality has been included. First, the means to access to the mental state of an individual agent. Each agent is identified with its local id in all cases.

```
MentalStateManager msm = MSMRepository.getInstance().get(
"MY_AGENT_ID");
```

The MSMRepository contains references to all Mental State Manager compoments of agents launched within the same JVM. The method *get* will return a null if the desired agent has not registered itself, yet. To prevent this, there is another primitive which waits 10 seconds for an agent to register its mental state manager.

```
MentalStateManager msm = MSMRepository.getInstance().waitFor(
"MY_AGENT_ID");
```

The call to *waitFor* method will cause the caller to wait until the callee receives a reference to the desired mental state manager. The wait has a timeout of 10 seconds, after which an exception will be triggered. Having access to this Mental State manager enables the test to insert, remove, delete, or query part of the mental state of the agent.

A similar functionality is provided to access the Mental State Processor of the agent. As in the previous case, a waitFor and a get method permits to obtain the reference to the Mental State Processor of each agent.

```
MentalStateProcessor msp1 = MSPRepository.getInstance().
waitFor( "MY_AGENT_ID1");
```

```
MentalStateProcessor msp2 = MSPRepository.getInstance().
get( "MY_AGENT_ID2");
```

After obtaining references to individual agents, it is recommendable to wait until the agent is fully initialized. The static method *waitForAgentInitialised* will do that for us if a reference to the mental state processor of the agent is passed as argument.

```
TestUtils.waitForAgentInitialised(msp2);
```

The previous set of instructions for obtaining the mental state processor, mental state manager, and wait for the initialization, have to be repeated for each agent to be included in the test.

Now the proper test starts. Like in JUnit, assert calls are used along the code. These calls frequently make a reference to some mental state manager, querying for elements of some type. For instance, if the mental state manager is msmA, and the developer looks for exactly one instance of a fact called FailAllParticipant2, then the following code would do the testing.

```
assertTrue("There should be once instance of
FailAllParticipant2",msmA.getMentalEntityByType("FailAllPartic
ipant2").size()==1);
```

Perhaps, the developer prefers to take a full snapshot to test what changed from this moment until the end of the execution. To verify this, the static snapShot method can be used.

TestUtils.snapShot(msm,"WRITE_HERE_A_NAME_FOR_THE_SNAPSHOT");

This method creates internally a copy of the mental state stored in msm and assigns the copy the name introduced as second argument.

Initially, each agent has not been started yet, though all should be initialized. This permits taking snapshots of the different mental state and analysed them to check they contain appropriate values.

When this initial inspection is finished, the following code will trigger all agents.

```
MainInteractionManager.goAutomatic();
TestUtils.doNothing(5000);
```

The first instruction will tell all agents to continue executing, enabling their capability to start choosing and launching tasks. At any moment, this capability can be switched off with `MainInteractionManager.goManual()`. As a result of the later, the execution of tasks would be stopped.

The second instruction just makes the test wait some time, enough for each agent to finish their work. The amount of time to wait depends on the estimation of each developer. Usually, one establishes a maximum response time and tunes the multi-agent system to not supere it. The time is measured in milliseconds.

The next step consists in inspecting again the mental state of individual agents.

The developer may want to check the current mental state of an agent against an initial snapshot. To do so, the following code can be used:

```
Vector<MentalEntity> differences =
TestUtils.compareCurrentStateAgainstSnapshot(msm,
"NAME_OF_THE_OLD_SNAPSHOT_HERE" );

assertTrue("Agent should remain the same and I found the
following differences "+differences,differences.size()==0);
```

The code assumes a previous snapshot of name NAME_OF_THE_OLD_SNAPSHOT_HERE. The *compare Current State Against Snapshot* method performs the comparisson between this mental state and the known one and enumerates all entities which are different or new. It does not test if a concrete entity was deleted, though.

To inspect concrete instances of the mental state, the getMentalEntityByType method can be used. Developers can check the number of instances or directly inspect the values.

```
assertTrue("There should be once instance of
FailAllParticipant2",msmA.getMentalEntityByType("FailAllPartic
ipant2").size()==1);
```

A different issue happens with interactions. An interaction in runtime is called conversation. To obtain conversations, a getMentalEntityByType can be used or the following code.

```
RuntimeConversation conv= TestUtils.
checkFirstConversation( msm,"LOCAL_AGENT_ID","CONVERSATION
STATE");
```

This code obtains the first conversation obtained from the mental state and checks its status. The status of a conversation can be aborted, initiated, or running. If there is no conversation or if its status is not the appropriate, an error is triggered. The inspected conversation is returned as a result, so further inspection is possible.

As told in previous sections, a task obtains information either from the mental state (non-conversational) or from an existing conversation (conversational). Therefore, to fully assess if an entity exist, conversations has to be checked as well.

```
TestUtils.checkNOExistenceMEWithinConv(conv,
"MENTAL_ENTITY_TYPE", "LOCAL_AGENT_ID");

TestUtils.checkNOExistenceMEWithinMS(msm,
"MENTAL_ENTITY_TYPE", "LOCAL_AGENT_ID");
```

The first instruction checks if no entity of type MENTAL_ENTITY_TYPE exists within the conversation. The second does the same within the mental state of the agent. Since there can be more than one instance entity of the same mental entity type, it may be necessary to check the exact number of instances.

```
TestUtils.checkExistenceMEWithinMS(msm, "MENTAL_ENTITY_TYPE",
"LOCAL_AGENT_ID",1);

TestUtils.checkExistenceMEWithinConv(conv,"MENTAL_ENTITY_TYPE"
, "LOCAL_AGENT_ID", 1);
```

The code above checks that exist exactly one instance of each corresponding mental entity type within the mental state or the conversation, respectively.

16.3 Launching the test

Each test or collection of tests can be launched from a JUnit capable development environment. If you choose this option, please, execute those tests allocated in the automatically generated output folder, which all start by the word Testing and are allcoated in the ingenias.testing package. If you chose the tests allocated in the permanent folder, which draw their name from the test entity name, probably you will get failures, since those tests do not include any initialization procedures.

If the test is launched from command line, the file build.xml, allocated into the project folder, will contain specialised tasks.

For instance, in the testTasks example there are the followign testing targets: **junitTestingTasks** and **alljunit**. The first refers to the deployment config referred in Figure 42. The second, refers to all defined junit test, though this case has only one.

Figure 42: Testing deployment

So, to launch all the tests, the following command can be used:

```
ant alljunit
```

If only a specific one is to be launch, the following command can be used:

```
ant  junitTestingTasks
```

Also, the referred classess inside the target definition in the build.xml can be run as junit in external development tools, like eclipse.

17 Troubleshooting

17.1 The interaction does not progress

An interaction in runtime, i.e., a conversation, does not progress mainly because the evidences to be sent do not exist yet.

Check the mental state of the involved agents against the definition of the interaction, concretely to the entities to be transferred. If these entities do not exist on the sender, perhaps there is the problem.

17.2 The task is not executed.

Review the logs to make sure the task is not executed. If necessary, clean the logs and start from scratch by pressing the **clear** button of the log panel. When a task is not executed, it is because there are not the required inputs in the mental state. If this is the case, the logs should show which ones are missing. Alternatively, the manual execution mode can be chosen. Within this mode, it is easier to inspect the mental state of the agent and check if the information the task requires is really missing.

17.3 The agent is not launched

Review the console output. Probably it will show an exception showing further details. Many times, it is a problem of launchin twice the same deployment without terminating a previous one.

17.4 Out of memory exception in the console

Besides the known causes for massive memory comsuption, most of them related with infinite loops or infinite recursive calls; perhaps it is simply a problem of a lack of prevision of the amount of memory the MAS requires. Try modifying the **memory** parameter, if defined. If it is not defined, create one. More details about this parameter can be found in section 12. After the modification takes place, regenerate the code and execute again.

Also, it may be a problem of generating too much output. Review the mechanisms the IAF uses to clean the mental state in section 6.3.

Try to define the conversations so that they can finish always one way or the other. This way, they can be removed automatically.

17.5 Messages do not arrive to the expected receiver

Check that the information is being transferred. The transfer will not occur until the sender produces this information in its mental state or within the conversation

17.6 The task does not finish, hence, the agent gets stuck

Review the code of the task. It may have fallen in an infinite loop or similar.

17.7 I need to allocate methods within the task for my code

This simply cannot be done, at least if the code generation capabilities are used. Any modification of the code outside the assigned area will be deleted.

If it is required to allocate additional methods, define an application and connect it to the task with an WFUses relationship. Alternatively, define static public methods in a separated class.

17.8 When the task accesses the information contained within a fact, it does not contain the information I expected

The best approach to this problem is to track down which tasks did produce this information and what happened to its different attributes since its creation. To get this information, the DebugUtils.printStack static method can be used. If the developer

supplies a mental entity extending a RuntimeFact, it is possible to print in the error output what has hapened to that entity along its live.

17.9 The MS gets crowded with entities.

Check that all asserted entities can in fact be consumed by agent tasks. It may be the case, the new information cannot be understood by existing tasks.

Mechanisms devised by the IAF help to prevent this kind of situations. Check the section dedicated to the mental state of the agent, section 6.

18 Figure Index