
AIMMS Multi Agent and Web Services User's Guide - Tutorial

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit www.aimms.com

Copyright © 1993–2011 by Paragon Decision Technology B.V. All rights reserved.

Paragon Decision Technology B.V.	Paragon Decision Technology Inc.	Paragon Decision Technology Pte.
Schipholweg 1	500 108th Avenue NE	Ltd.
2034 LS Haarlem	Ste. # 1085	80 Raffles Place
The Netherlands	Bellevue, WA 98004	UOB Plaza 1, Level 36-01
Tel.: +31 23 5511512	USA	Singapore 048624
Fax: +31 23 5511517	Tel.: +1 425 458 4024	Tel.: +65 9640 4182
	Fax: +1 425 458 4025	

Email: info@aimms.com
WWW: www.aimms.com

AIMMS is a registered trademark of Paragon Decision Technology B.V. IBM ILOG CPLEX and sc CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. KNITRO is a registered trademark of Ziena Optimization, Inc. XPRESS-MP is a registered trademark of FICO Fair Isaac Corporation. MOSEK is a registered trademark of Mosek ApS. WINDOWS and EXCEL are registered trademarks of Microsoft Corporation. \TeX , \LaTeX , and $\AMS-\LaTeX$ are trademarks of the American Mathematical Society. LUCIDA is a registered trademark of Bigelow & Holmes Inc. ACROBAT is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of Paragon Decision Technology B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Paragon Decision Technology B.V.

Paragon Decision Technology B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Paragon Decision Technology B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.

This documentation was typeset by Paragon Decision Technology B.V. using \LaTeX and the LUCIDA font family.

Chapter 2

Tutorial

In this chapter there is a small but complete example of a multi-agent application to be built using AIMMS. Following the description of the underlying problem, you will need to specify an agent community, the agent setup for each of four different agent roles, and the procedures that make up the communication between the agents.

This chapter

This tutorial focuses purely on the multi-agent features of AIMMS. Some basic AIMMS knowledge is required but, contrary to what you might expect, the example does not include any optimization. This is avoided to keep the example as simple as possible, allowing you to focus on learning the multi-agent concepts in AIMMS. If you lack the necessary AIMMS knowledge you are advised to read one of the AIMMS tutorials that come with the AIMMS distribution package.

Requirements

In order to make this tutorial, or run the completed implementation provided with the AIMMS distribution, you need a license that allows you to simultaneously run the indicated amount of AIMMS sessions. If you have a network license with a sufficient amount of free AIMMS licenses, you can use these. If you only have a single AIMMS license, but still want to experiment with the AIMMS multi-agent technology, for instance, by making this tutorial, Paragon will provide you, on request, with a temporary AIMMS license capable of running the tutorial example.

Licensing

2.1 Describing the agent application

The example in this chapter concerns the simulation of a help desk. Assume that the help desk center has *priority customers* who request help through their dedicated *account managers*. Each account manager passes the request to the first available *worker* to handle the request in its entirety. The overall performance of the help desk is monitored by a single *controller*. Figure 2.1 illustrates the connections between individual customers, account managers and workers.

A help desk simulation ...

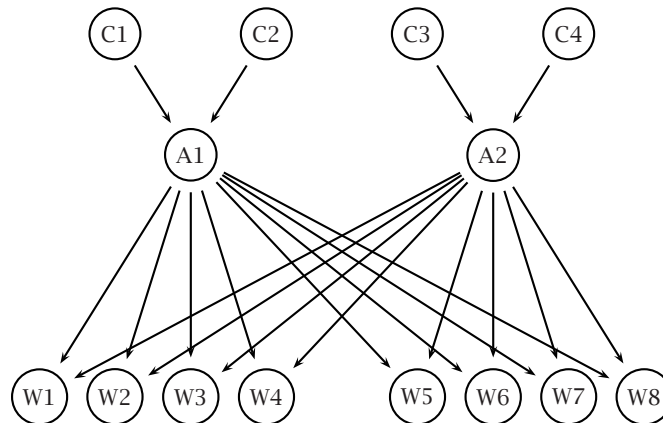


Figure 2.1: Connections between individual customers (C), account managers (A) and workers (W)

The simulation is implemented as a multi-agent application with the following agent roles

- customer,
- account manager,
- worker, and
- controller.

*... as a
multi-agent
application*

For convenience, all agents, other than the controller, are started by the controller in the example. In reality, however, agents could also begin, and log on to the community, independently. Using the functionality described later in Chapter 3, the example could easily be modified to allow this. The number of agents that are started is kept small, but this can be enlarged if you wish so. In total, there will be 4 customers, 2 account managers, 8 workers and 1 controller.

During the simulation, a customer agent sends messages requesting help to its designated account manager. The time interval between requests varies over time, and is simulated using an exponential distribution. The corresponding distribution parameter value is provided as input by the controller.

Customer

Once a request for help reaches an account manager, this request is forwarded to the first available worker. If no worker is available, the request is queued. Whenever a worker becomes available, any account manager with a nonempty queue requests exclusive access to this worker. If exclusive access is granted by the worker, the account manager forwards the first request on his queue to this worker. If exclusive access is denied, the account manager waits until another worker becomes available again.

*Account
manager*

A worker is able to handle a help request forwarded by any account manager. Workers notify all account managers whenever they become available after having completed a request. Whenever a worker receives a request for exclusive access from an account manager, and is still available, that access is granted. The time required to complete the help request by a worker is assumed to be uniformly distributed. The corresponding distribution parameter values are provided as input by the controller.

Worker

The controller is a special agent that manages the community by

The controller

- starting and stopping agents,
- providing input parameters to agents, and
- gathering performance statistics from agents.

Once the simulation is started, the controller can influence the performance of individual agents by modifying their distribution parameters.

2.1.1 Message types and semantics

The functionality described in the previous section can be implemented by including several message types in the agent application. In the paragraphs below, the message types sent, by all types of agents, will be discussed in more detail. An agent sends a message by calling a corresponding send procedure, which relays the message to the underlying agent technology. Here, all relevant message data are encapsulated, and the message is passed on to the receiving agent. As we will see later, AIMMS can automatically add the appropriate send procedures to an AIMMS project implementing a particular agent role.

Sending messages

Messages sent from one agent to another agent must be either filtered or handled by the receiving agent. The underlying agent technology ensures that any incoming message is correctly filtered or handled. This is accomplished by calling the designated filtering or handling procedure, within the AIMMS project of the receiving agent, whenever a new message arrives. When setting up an AIMMS project to implement a particular agent role, AIMMS can automatically add skeleton filtering and handling procedures to the project. You, as an agent developer, can then modify these skeleton procedures to take the actions that you want.

Handling messages

Customer agents send RequestHelp messages to their designated account managers to simulate a help request. For the sake of simplicity, it is assumed that these messages do not carry any extra data arguments. Account managers use a filter procedure for incoming RequestHelp messages, in which they

*The Request-
Help message type*

- forward an incoming message to an available worker, or
- if no worker is available, place the incoming message in a queue.

The worker agents have a handler procedure for incoming RequestHelp messages, in which they

- execute the help request according to an exponential distribution, and then
- notify the account managers that they are available again.

Whenever a worker agent finishes a request for help and becomes available again, he sends a NotificationAvailability message to all account manager agents. These messages do not carry any additional data arguments. The account managers have a handler procedure for incoming NotificationAvailability messages, in which they

The NotificationAvailability message type

- check their own queue of help requests,
- request exclusive access when this queue is nonempty, and
- forward the first message when access is granted.

To guarantee that a worker agent will honor a forwarded RequestHelp message, account managers will send a RequestExclusiveAccess message prior to actually forwarding the RequestHelp message. RequestExclusiveAccess messages are priority messages, enforcing an immediate response, and have one scalar output argument registering whether or not exclusive access has been granted. This prevents multiple account managers forwarding help requests simultaneously, of which only the one arriving first will be handled immediately. Worker agents have a handler procedure for incoming RequestExclusiveAccess messages, in which they

The Request-ExclusiveAccess message type

- verify their availability,
- if available, grant exclusive access to the account manager, and
- notify the account manager of the result.

With a real help desk, customers request help as the need arises, and help desk personnel have to perform actual work to handle these requests. In this simulation, however, customer agents will generate RequestHelp messages at random points in time according to a given distribution, and worker agents have to spend a random amount of time doing nothing when simulating handling a RequestHelp message. Although such behavior could be implemented using AIMMS' built-in Delay function, this function would completely block the AIMMS execution, which is undesirable in an otherwise asynchronous application. It is important for an agent to be non-blocking, because this allows AIMMS to deal with incoming priority messages, or, if necessary, update the user interface of the agent.

Simulation requires delay

An alternative, non-blocking, implementation can be achieved by the introduction of two auxiliary delayed message types, `DelayedGenerateRequest` and `DelayedFinishRequest`, which customer and worker agents respectively can send to themselves. When sending a message from within an agent project, AIMMS allows you to specify a delay time, which will cause AIMMS to wait for the specified amount of time before actually sending the message. By using this feature whenever a `DelayedGenerateRequest` or `DelayedFinishRequest` message is sent, the required random delay times for customer and worker agents are achieved in a non-blocking manner.

*Auxiliary
delayed
message types
...*

Although we have spent two paragraphs explaining the delayed message types, you should realize that they are only a trick to bridge, in a non-blocking manner, a time interval when no real work needs to be executed during the help desk simulation. If you are writing an agent application along similar lines, you probably will have no need for this trick, because actual execution, consisting of multiple interruptible statements, is necessary when handling each of the message types present in your application.

... are a trick

The communication pattern between the various agent types is illustrated in Figure 2.2. The solid arrows correspond to the message types described above. The dashed arrows correspond to auxiliary message types initiated by the controller to initialize, start and stop the simulation. These auxiliary message types are discussed in Section 2.1.2.

*Graphical
display*

2.1.2 Tasks of the controller agent

The first task of the controller is to startup all the other agents in the community. The controller will wait until every agent has started successfully to ensure that every agent is available to take part in the simulation.

*The controller
starts agents ...*

Once all the agents are present, the controller sends a message of type `InitializeAgentSettings` to initialize such agent parameters as

*... starts the
simulation*

- the designated account manager and average number of requests for customer agents, and
- the minimum and maximum work duration for worker agents.

The `InitializeAgentSettings` message includes the property that an automatic response is sent back to the controller once the message has been handled. The controller waits for all these responses before actually starting the simulation, by sending a `StartSimulation` message to all customer agents.

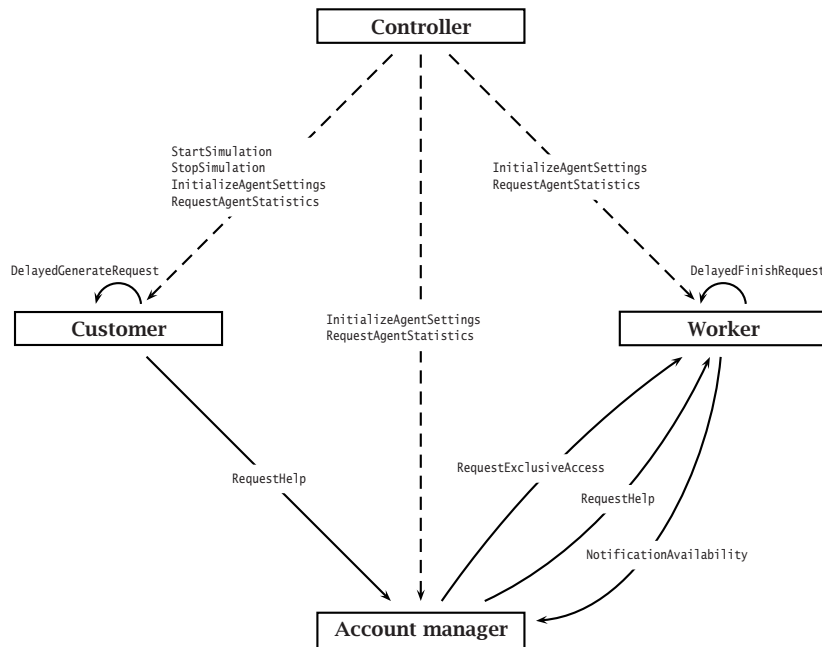


Figure 2.2: Message communication between agent roles

During the simulation, the controller can modify the parameter values of selected customers and workers by sending similar initialization messages again. In addition, the controller can send priority messages of type RequestAgentStatistics to obtain statistics while the simulation is still running. Examples of such simulation statistics are

... monitors

- the number of requests generated by a particular customer agent,
- the queue size and the number of forwarded messages of a particular account manager, and
- the number of jobs, and the total time spent on them, executed by a particular worker.

The controller can stop the simulation by sending the message of type StopSimulation to all customer agents. The effect of this message is that the customers no longer generate new help requests, but do remain present. Note that account managers and workers are not effected by this message, and will remain operational until all help requests have been handled.

... stops the simulation

Once the simulation has been stopped, the controller can terminate all other agents in the community, even though they may still be handling outstanding requests. In general, any agent that starts other agents is also responsible for terminating them.

*... terminates
the agents*

2.2 Building the agent application

In this section all the relevant issues that play a role in the implementation of the multi-agent application of this tutorial are described. Section 2.2.1 discusses how an AIMMS project can be prepared for becoming an agent in an AIMMS-based multi-agent application, and how the corresponding agent community can be set up. Section 2.2.2 illustrates how AIMMS can add all send, filter and handler procedures for a particular agent role to an agent project. Section 2.2.4 provides the actual implementation of all the message handler procedures that implement the actual help desk simulation as described in Section 2.1.1. Finally, in Sections 2.2.5 and 2.2.6, all the controller-related tasks, messages and handlers are discussed.

This section

2.2.1 Agent preparation and community setup

For any AIMMS project that is to act as an agent in an AIMMS-based multi-agent community, some preparatory steps are required. These steps include

*The agent
preparation
process*

- installing the agent module,
- specifying the community setup file, this contains the description of all agent roles and message types, and
- performing the agent setup for one or more agent roles.

A *community setup* file contains the shared information to be used by all agents within a single multi-agent community. This information includes

*The community
setup*

- all available agent roles,
- all available message types and their arguments, and
- a map describing which message types can be sent and received by which agent roles.

The community setup file should ideally be under the control of a single person.

Although AIMMS allows the implementation of several agent roles within the same AIMMS project, typically, you will create an AIMMS project for each agent role. Such a project can then be started for every instance of the corresponding agent role. Recall that an agent project can be opened more than once, only if it is opened in end-user mode. In end-user mode you can run the project, but cannot make any changes to it. When starting a new agent from within another

*One AIMMS
project per
agent role*

agent using the `StartRemoteProject` command, the new agent is always opened in end-user mode.

In this tutorial we will build four separate AIMMS projects, one for each agent role. Each project is stored in a subdirectory of the directory `C:\AgentTutorial\`, which was created manually. Using the **New Project** wizard, the controller agent project can now be created following the information shown in Figure 2.3.

Creating a new project

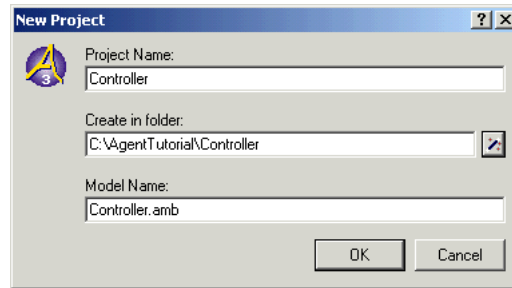


Figure 2.3: Creating the controller agent project

To install the multi-agent module

- ▶ select the **Install Agent Module** command from the **Settings-Multi Agent** menu.

Installing the multi-agent module

This will import a predefined multi-agent module into your model. After installing the agent module you should be able to verify the existence of the newly created agent module in the model tree, as illustrated in Figure 2.4.

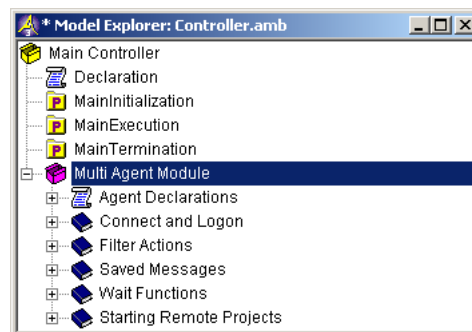


Figure 2.4: The model tree with the newly installed agent module

The community setup file is typically shared among all agents in the community. To specify the location of the community setup file perform the following steps

Specifying the community setup file

- ▶ select the **Select Community Setup File...** command from the **Settings-Multi Agent** menu, and
- ▶ specify `..\CommunitySetup.cfg` as the name of the community setup file (see Figure 2.5).

If you specify a non-existing community setup file, AIMMS will ask whether you want to create a new community setup file with the specified name. Typically, this will happen when you start building a new multi-agent application. All other agent projects in the multi-agent application should select the same community file in order to share the information specified in it.

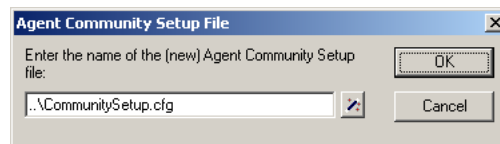


Figure 2.5: Specifying the location of the community setup file

Since setting up a correctly working multi-agent community is not a straightforward task, the community setup file should preferably be under the control of a single person. To prevent other persons from modifying the community setup file, you can password protect it through the **Settings-Multi Agent-Community Setup Password** command.

Protecting the community setup file

Before continuing with the agent role-specific initialization of the project, you have to specify the contents of the community setup file. Since the community setup information is shared among all agents, this data can be specified from within any agent project in the multi-agent application. Changes made to the community setup file from within in one agent project will be automatically propagated to the other agents. To open the **Multi Agent Community Setup** dialog box

Community setup

- ▶ select the **Community Setup ...** command from the **Settings-Multi Agent** menu.

The first step, in setting up a multi-agent community, is to specify all available agent roles. To specify all the agent roles that play a role in this tutorial

Agent roles

- ▶ select the **Agent Roles** tab in the **Multi Agent Community Setup** dialog box, and
- ▶ use the **Add** button (or the **Insert** key on the keyboard) to add the four agent roles introduced in Section 2.1 (see Figure 2.6).

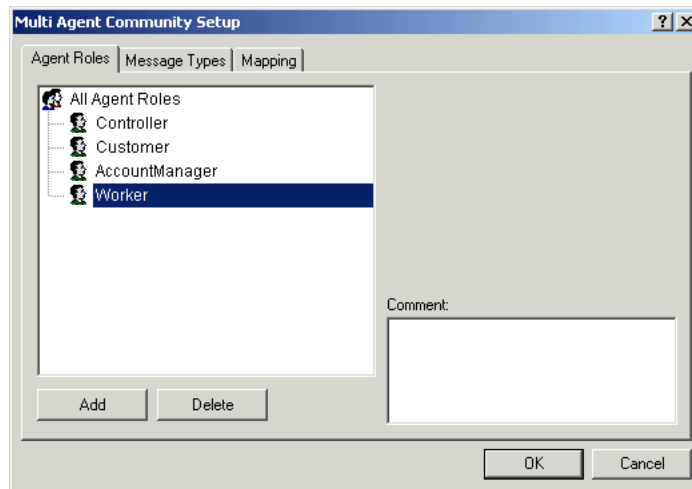


Figure 2.6: Specifying all the agent roles

If you are designing an agent application yourself, you can add explanatory remarks about a particular agent role in the **Comment** field.

The second step, in setting up the multi-agent community, is to declare the message types that were displayed in Figure 2.2. To create the RequestHelp message type (introduced in Section 2.1.1):

Creating message types

- ▶ select the **Message Types** tab in the **Multi Agent Community Setup** dialog box, and
- ▶ use the **Add** button (or the **Insert** key) to add the RequestHelp message type (see Figure 2.7).

As illustrated in Figure 2.7, AIMMS supports several options for messages sent between agents.

Message options

- By declaring a message type as a “*Priority Message*”, AIMMS will handle incoming messages of this type at the earliest possible moment, possibly during the execution of a message handler of a non-priority message.
- By checking the “*Sends Response when Handled*” option, AIMMS will send a response message back after the message has been handled by the receiving agent. In the sending agent, you can wait for these response messages through the function `WaitForHandledMessage`.
- By checking the “*Adds Elements to Domain Sets*” option, AIMMS will automatically add elements to domain sets, whenever multidimensional message arguments refer to set elements that are not in the corresponding domain sets of the receiving agent. Without this option, AIMMS will generate a runtime error if a domain element is not present.

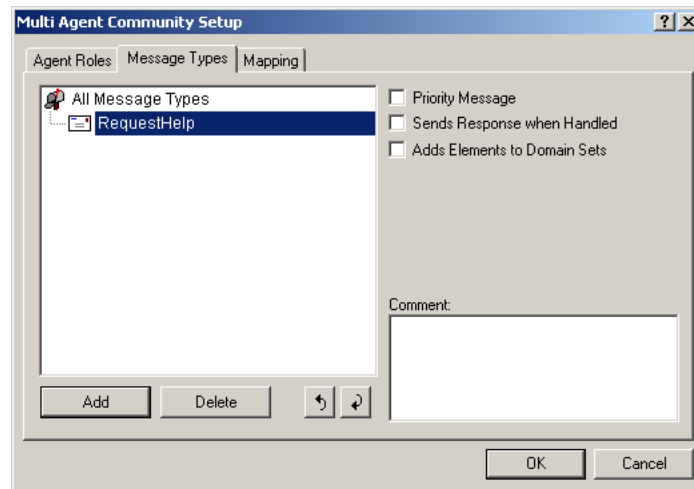




Figure 2.7: Declaring the RequestHelp message type

None of these options apply to the RequestHelp message type.

The RequestExclusiveAccess message, described in Section 2.1.1, has a single output argument IsGranted to indicate whether exclusive access has been granted by the receiving working agent to the requesting account manager. To ensure timely delivery of the message, it should also be declared as a priority message. To add the RequestExclusiveAccess message type to the multi-agent community setup, you have to perform the following actions

Message types with arguments

- ▶ on the **Message Types** tab, add a new message type with name RequestExclusiveAccess,
- ▶ check the **Priority Message** checkbox,
- ▶ double click the letter icon  to open a new level below the message (the icon will change to ) ,
- ▶ press the **Add** button (or the **Insert** key) to insert a new double-valued argument IsGranted (see Figure 2.8), and
- ▶ select the **Output** radio button for the IsGranted argument.

To complete the main set of message types necessary to run the help desk simulation, you should now add the NotificationAvailability message type, discussed in Section 2.1.1, as well as the DelayedGenerateRequest and DelayedFinishRequest message types, all without any arguments. After completion, the **Message Types** tab should now contain the five main message types as illustrated in Figure 2.8.

The message types so far

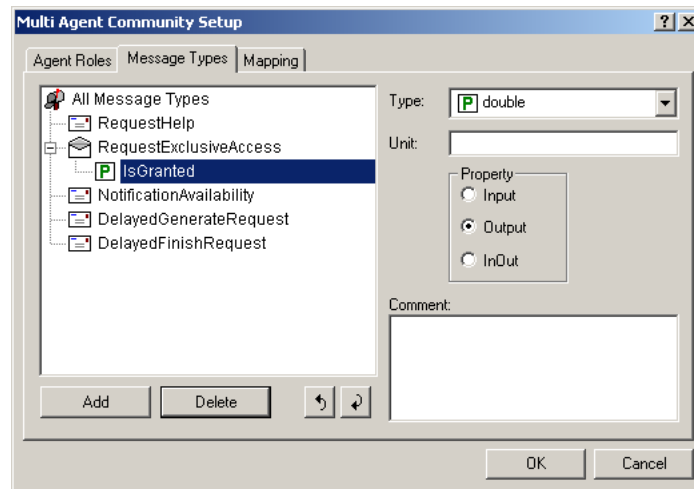


Figure 2.8: The main message types

The message types introduced so far have all been related to running the actual help desk simulation. We still have to declare the controller-related messages discussed in Section 2.1.2, and represented by the dashed arrows in Figure 2.2.

Controller-related message types

The `InitializeAgentSettings` message, discussed in Section 2.1.2, is responsible for initializing the agent parameters of the customer, the account manager and the worker agents. For example, the individual customer agents differ in the account manager to which they have been assigned, and in the rate with which help requests are generated. By sending this information from the controller to the customer agents, rather than letting this information be part of the customer project itself, all customer projects are identical and can therefore be shared among all individual customer agents. A similar argument holds for the individual account managers and worker agents.

Initialization messages

To ensure that the help desk simulation only starts after all the agents have been initialized correctly, the initialization messages above can be marked as *Sends response when handled*. After execution by the handler, the receiving agent will automatically send a special *handled* message back to the controller agent. As we will see later on, the controller will be able to postpone its own execution until all the other agents have been initialized by using the function `WaitForMultipleHandledMessages`.

Handled response messages

The `InitializeAgentSettings` message has two arguments. The `AnAgent` input argument is an element parameter into the set `AllAgents`, and can be used to denote the account manager, assigned to a customer agent, that is responsible for handling the customer's requests. The 1-dimensional input argument `AgentParameters(p)`, where `p` is an index into some unspecified set intended to

Message arguments

hold the names of possible all agent parameters, can be used to initialize various agent parameters, such as the average number of requests by customer agents, or the minimum and maximum work duration of worker agents. Include the `InitializeAgentSettings` message in the community setup as shown in Figure 2.9.

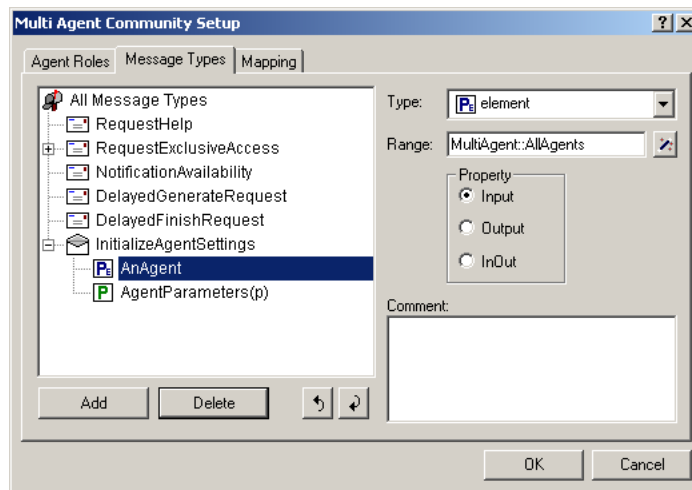


Figure 2.9: The `InitializeAgentSettings` message type

To start and stop the simulation, the controller agent will use the two message types `StartSimulation` and `StopSimulation` which will be sent to all customer agents as discussed in Section 2.1.2. These two message types do not have any arguments. You can now extend the multi-agent community setup with these two message types as illustrated in Figure 2.10.

Starting and stopping the simulation

To investigate the overall behavior of the simulation, each agent instance has to keep track of its individual behavior and should store the relevant statistical data in one or more model identifiers. To be able to investigate and compare these statistics, the controller agent can send `RequestAgentStatistics` messages to all agents to retrieve the statistical information collected so far. The gathered statistics will differ for each agent role:

Gathering status information

- customer agents keep track of the total number of requests generated,
- account manager agents keep track of the number of requests forwarded to workers and their current queue size, and
- worker agents keep track of the number of requests handled and the amount of time spent processing requests.

To pass these statistics, the `RequestAgentStatistics` message has a single 1-dimensional output argument `AgentStatistics(s)`, where `s` is an index into some unspecified set holding the names of all types of statistics gathered.

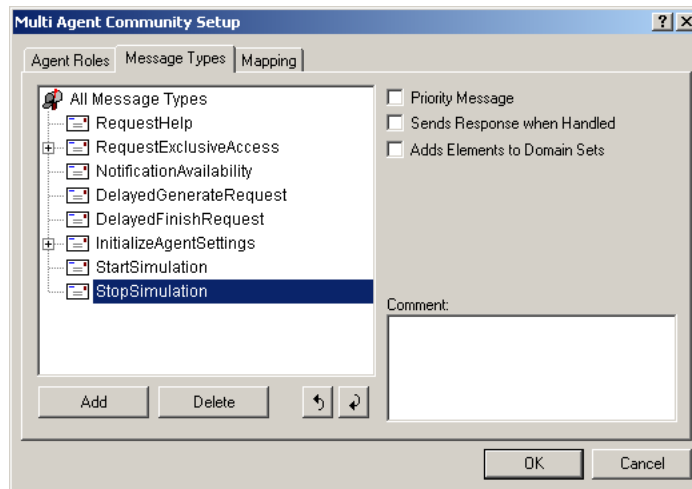


Figure 2.10: The message types that control the simulation

To ensure that the individual agents respond to status requests immediately, the status request messages are declared *priority messages*. Include the RequestAgentStatistics message type in the community setup as illustrated in Figure 2.11. Note, that this message type also has the “Adds Elements to Do-

main Sets messages

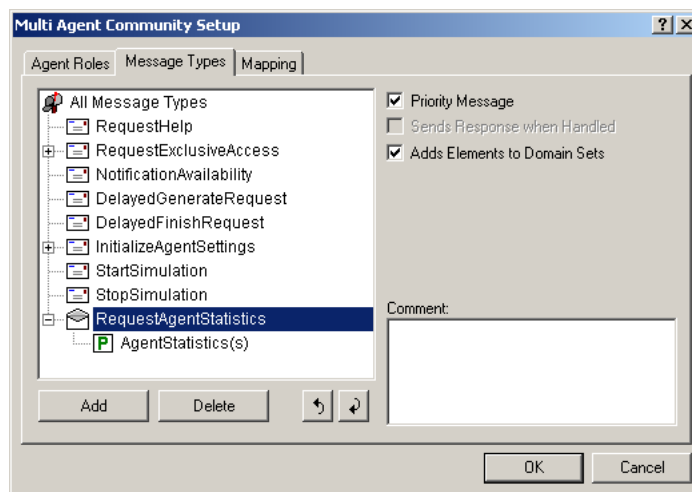


Figure 2.11: The RequestAgentStatistics message type

main Sets” option checked. In this manner, new statistics gathered by a particular agent type will be automatically added to the set containing all statistics items in the controller agent.

The final step in specifying a community setup, is to specify the message mapping. Through message mapping you can determine which message types can be sent and received by which agent roles.

Specifying a message mapping

Since the help request is generated by the customer, sent to the account manager and then forwarded to the worker, both the customer and the account manager should be able to send messages of this type, while both the account manager and the worker should be able to receive RequestHelp messages. To specify the message mapping for the RequestHelp message type

Message mapping for RequestHelp message

- ▶ open the **Mapping** tab of the **Multi Agent Community Setup** dialog box,
- ▶ select the RequestHelp message from the drop-down listbox,
- ▶ in the **From Agents** listbox on the left, select both the Customer and the AccountManager roles, and
- ▶ in the **To Agents** listbox on the right, select both the AccountManager and the Worker roles (see Figure 2.12).

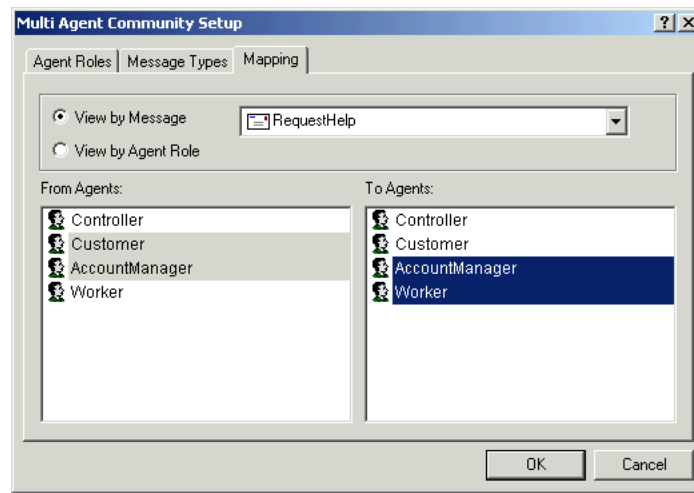


Figure 2.12: Specifying the message mapping for the RequestHelp message type

You can now complete the message mappings for the remaining message types bearing in mind that

Completing the message mappings

- RequestExclusiveAccess messages are sent by an account manager to a worker,
- NotificationAvailability messages are sent from a worker to an account manager,
- DelayedGenerateRequest messages are sent from a customer to a customer,
- DelayedFinishRequest messages are sent from a worker to a worker,

- StartSimulation and StopSimulation messages are sent by the controller to a customer, and
- InitializeAgentSettings and RequestAgentStatistics messages are sent by the controller to a customer, an account manager, or a worker.

You have now completed the community setup for the help desk simulation application, and can leave the **Multi-Agent Community Setup** dialog box by pressing the **OK** button.

Finishing the community setup

Before implementing the individual agent roles, you need to create new AIMMS projects for these agent roles (i.e. the customer, the account manager, and the worker agent role). It is suggested that you set up these projects in a similar directory structure as shown in Figure 2.13.

Creating the other agents' projects

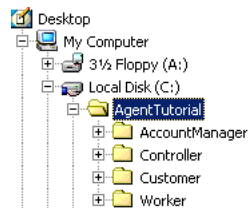



Figure 2.13: The proposed directory structure for all AIMMS projects in the community

Following the directory structure outlined in Figure 2.13, you can create the other agent projects similar to that illustrated in the beginning of this section for the controller project, by

Sharing the community setup file ...

- ▶ installing the multi-agent module in the newly created projects, and
- ▶ specifying `..\CommunitySetup.cfg` as the location of the community setup file.

If you use the wizard button  to select the existing community setup file, you will be asked whether you want to use the relative path to this file instead of the full path. You should use a relative path to an existing community setup file whenever possible, since the application will then still run when the four AIMMS projects are moved to a different location on your computer.

... by using a relative path

When developing a multi-agent application, you will often find yourself making changes to several AIMMS projects at once. To allow you to move quickly between several agent projects, it is very convenient to open all agent projects that simultaneously play a role in the community. It may also be convenient to set the *Project Title* option of each agent project to the corresponding agent type. This will help you find a particular agent project quickly in the Windows task bar.

Opening all AIMMS projects simultaneously

2.2.2 Agent setup

Having completed the community setup, it is now time to perform an agent setup for all the agent projects. In the agent setup phase you will *Agent setup*

- assign an agent role to an agent project,
- let AIMMS add send procedures to the project for all message types that can be sent by the selected agent role, and
- let AIMMS add filter and/or handler skeleton procedures to the project for all message types that can be received by the selected agent role.

In this section, we will guide you explicitly through the agent setup for the account manager role, which contains send, filter, and handler procedures. The agent setup for the other agent roles is similar, and is left as an exercise.

Before specifying the agent-related send, filter and handler procedures for an agent project, you should first specify its agent role. The multi-agent technology in AIMMS allows a single AIMMS project to perform several roles at once. However, in this example, each AIMMS project will only implement a single agent role. To specify the agent role for the account manager project *Specifying the agent role ...*

- ▶ select the account manager project,
- ▶ select the **Agent Setup...** command from the **Settings-Multi Agent** menu,
- ▶ select the **Agent Roles** tab in the **Multi Agent Implementation** dialog box,
- ▶ select the “AccountManager” role to be the role of the account manager agent (see Figure 2.14), and
- ▶ press the **OK** button.

Similarly, you can specify the agent roles for the controller, the customer and the worker projects.

A send procedure is an AIMMS procedure that is called in order to actually send a message to another agent. In the **Agent Role Setup** dialog box, AIMMS can automatically create these send procedures for any message type that you specify. The account manager agent only sends `RequestExclusiveAccess` messages directly. To generate the corresponding send procedure in the account manager project *Specifying a send procedure*

- ▶ select the account manager project,
- ▶ select the **Agent Setup...** command from the **Settings-Multi Agent** menu,
- ▶ select the **Sending** tab in the **Agent Role Setup** dialog box (see Figure 2.15),
- ▶ select the `RequestExclusiveAccess` message,
- ▶ check the **Send Procedure** checkbox, and
- ▶ press the **OK** button.

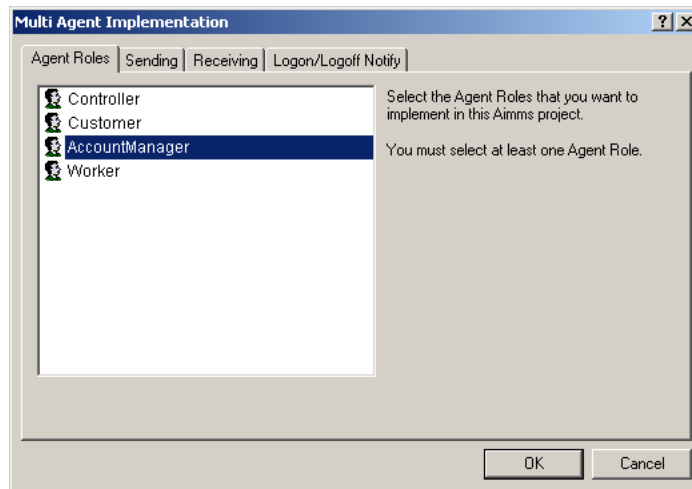


Figure 2.14: Specifying the agent role for the account manager agent

When you check the **Send Procedure** checkbox, AIMMS automatically proposes the name `Send_RequestExclusiveAccess` as the name of the send procedure, as illustrated in Figure 2.15. When you close the dialog box, AIMMS will check and inform you that the procedure does not yet exist in your model, and will ask whether you want to have this procedure created within your model. On pressing the **OK** button, AIMMS will generate an external AIMMS procedure that passes the request on to the multi-agent layer of AIMMS.

Automatically generated

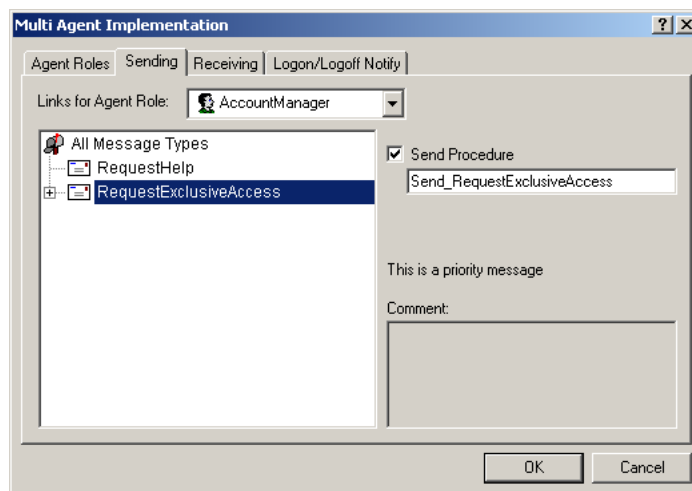


Figure 2.15: Specifying the send procedure for the RequestExclusiveAccess message

When adding the send procedure to your model, AIMMS will add a Multi Agent Send Procedures section to the model tree of the account manager project, and create the Send_RequestExclusiveAccess send procedure directly below it. The resulting model tree, for the account manager project, is displayed in Figure 2.16,

The send procedure in the model tree ...

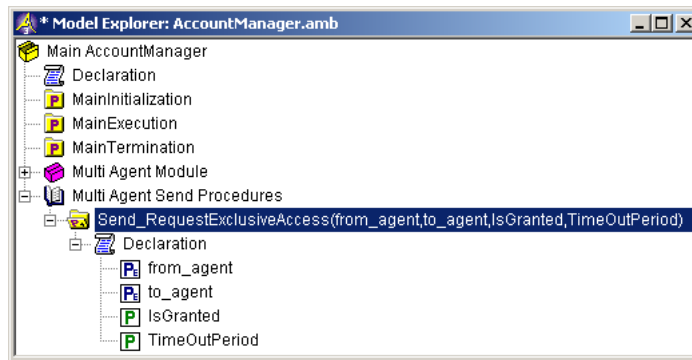


Figure 2.16: The send procedure for the RequestExclusiveAccess message

When creating the send procedure for a particular message type, AIMMS will examine the message arguments and properties, and, based on this information, create an external procedure with the appropriate arguments and attributes. The attribute form of the send procedure for the RequestExclusiveAccess is illustrated in Figure 2.17. You should not try to change manually any of the attributes of this procedure. If there are changes to be made in the message type, you should open the **Agent Role Setup** dialog box, and let AIMMS modify or recreate the procedure.

... and its attributes

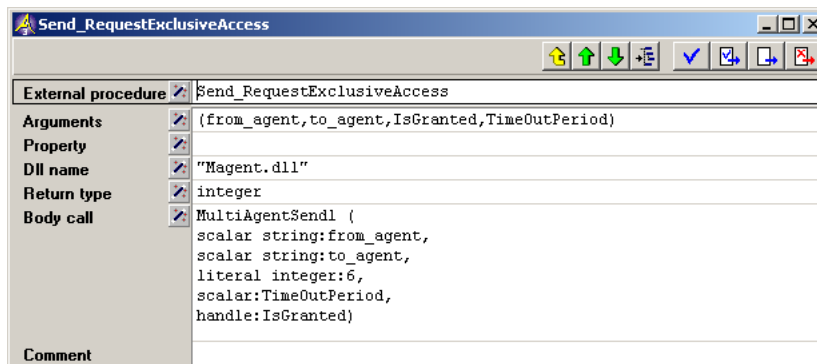


Figure 2.17: The attribute form of the send procedure for the RequestExclusiveAccess message

The community setup allows RequestHelp messages to be sent from an account manager to a worker agent. This is illustrated by the presence of the RequestHelp message type in Figure 2.15. However, since an account manager agent will never directly send a RequestHelp message to a worker agent, but only forward RequestHelp messages that have been received from customer agents, the account manager project does not require a send procedure for the RequestHelp message type.

No send procedure for RequestHelp messages

Since an account manager agent only forwards RequestHelp messages but never handles them, the account manager project only needs a filter procedure for the RequestHelp message type. Within a message filter you cannot make references to message arguments, but you can forward, store, or delete received messages. For an account manager agent, this provides sufficient functionality to deal with RequestHelp messages appropriately. To create a message filter for the RequestHelp message in the account manager agent:

Specifying a message filter procedure

- ▶ select the **Agent Setup...** command from the **Settings-Multi Agent** menu,
- ▶ select the **Receiving** tab in the **Agent Role Setup** dialog box (see Figure 2.18),
- ▶ select the RequestHelp message,
- ▶ check the **Message Filter** checkbox, and
- ▶ press the **OK** button.

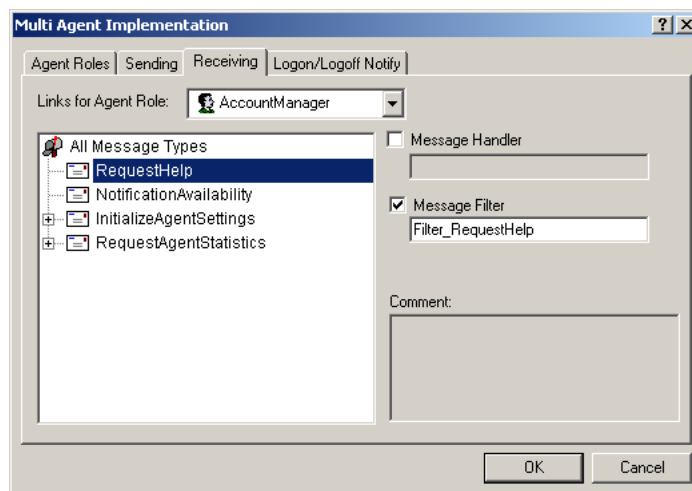


Figure 2.18: Creating a filter procedure for the RequestHelp message

As was the case with send procedures, AIMMS can automatically create filter (and also handler) procedures that do not yet exist in your model. However, the filter and handler procedures created by AIMMS are empty skeleton procedures, for which you have to specify the bodies yourself to match the intended semantics. When a message, of a particular message type, arrives at an agent session, the multi-agent layer will automatically call the corresponding filter and/or handler procedure.

Creating filter and handler procedures

The resulting model tree for the account manager project is displayed in Figure 2.19. We will specify the contents of the body of the Filter_RequestHelp procedure in Section 2.2.4.

The filter procedure in the model tree

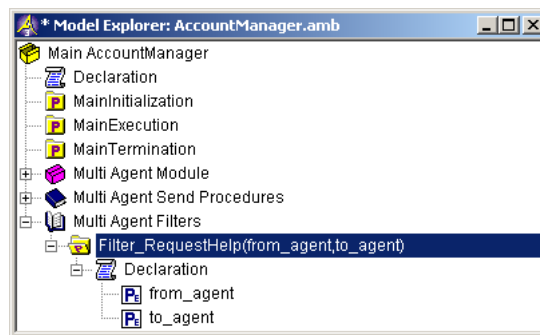


Figure 2.19: The filter procedure for the RequestHelp message

An account manager agent must be able to handle NotificationAvailability messages sent by worker agents, as well as the InitializeAgentSettings and the RequestAgentStatisticsData messages sent by the controller agent. Adding a message handler Handler_NotificationAvailability for the NotificationAvailability message is similar to adding the filter procedure for RequestHelp messages, and is left as an exercise. The InitializeAgentSettings and RequestAgentStatisticsData message types, however, have one-dimensional arguments which have to be dealt with appropriately.

Specifying handler procedures

When a message type has associated message arguments, these arguments will normally translate into procedure arguments with the same name in both the send and handler procedures (recall that filter procedures do not provide access to the message arguments). For multidimensional arguments, however, we have to link the formal and unspecified domain indices as declared in the message type declaration of the community setup, to specific sets in the agent project. These formal indices are left unspecified during the community setup for a good reason: the corresponding domain sets may be different for each agent project.

Dealing with message arguments

When linking a formal index to a set in an agent project, there are basically two choices:

- link the index to an existing global set within the agent project, or
- link the index to a local set within the send or handler procedure.

Whenever the formal index name already exists as an index into a global set, AIMMS will automatically assume that the formal index name refers to the global index. In all other cases, AIMMS will ask you to specify the set with which the formal index name must be associated. If you specify a non-existing set name, AIMMS will ask whether you want to create the set using a local set declaration.

In this tutorial, we will always associate the formal index *p*, of the `AgentParameters(p)` argument of the `InitializeAgentSettings` message, with a global set `ParameterItems`, and the formal index *s*, of the `AgentStatistics(s)` argument of the `RequestAgentStatistics` message, with a global set `StatisticsItems`. You should add these sets to all four agent projects. To add these sets to the account manager project

- ▶ add the sets `ParameterItems` and `StatisticsItems` to the model tree as illustrated in Figure 2.21,
- ▶ add the index *p* to the `INDEX` attribute of the set `ParameterItems`, and
- ▶ add the index *s* to the `INDEX` attribute of the set `StatisticsItems`.

To specify the handler procedure for the `InitializeAgentSettings` message in the account manager project:

- ▶ select the **Agent Setup...** command from the **Settings-Multi Agent** menu,
- ▶ select the **Receiving** tab in the **Agent Role Setup** dialog box (see Figure 2.20),
- ▶ select the `InitializeAgentSettings` message,
- ▶ check the **Message Handler** checkbox, and
- ▶ press the **OK** button.

When creating the handler procedure, AIMMS will display a message arguments dialog box, in which you are allowed to modify the names of the procedure arguments to be created. In this instance, you should just press the **OK** button, after which AIMMS will add the handler procedure to your model.

After you have also added message handlers for the `NotificationAvailability` and `RequestAgentStatistics` messages, the model tree of the account manager project should resemble the tree illustrated in Figure 2.21.

Linking formal indices to sets

The sets ParameterItems and StatisticsItems

Specifying the handler procedure

Final account manager model tree

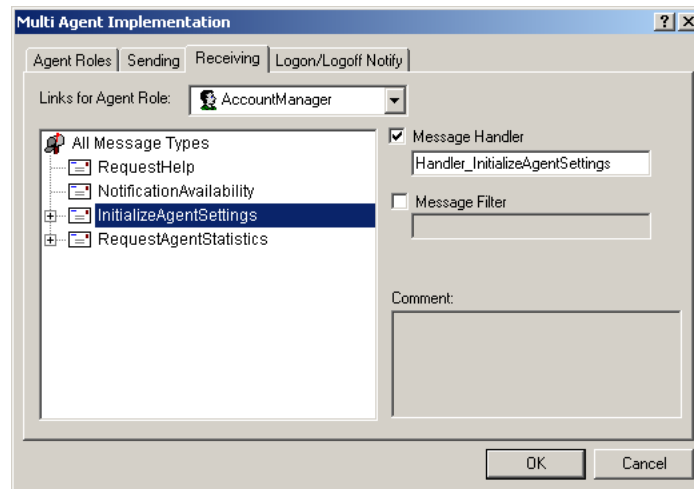


Figure 2.20: Specifying the handler procedure for the InitializeAgentSettings message

You should now create the message handlers and send procedures for the controller, customer and worker agent projects, in a similar way as for the account manager project. More specifically, you should add

Creating the remaining handlers and send procedures

- to the controller agent project:
 - send procedures for the
 - InitializeAgentSettings,
 - StartSimulation,
 - StopSimulation, and
 - RequestAgentStatistics message types.
- to the customer agent project:
 - send procedures for the
 - RequestHelp, and
 - DelayedGenerateRequest message types.
 - handler procedures for the
 - DelayedGenerateRequest,
 - InitializeAgentSettings,
 - StartSimulation,
 - StopSimulation, and
 - RequestAgentStatistics message types.
- to the worker agent project:
 - send procedures for the
 - NotificationAvailability, and
 - DelayedFinishRequest message types.
 - handler procedures for the
 - RequestHelp,
 - RequestExclusiveAccess,

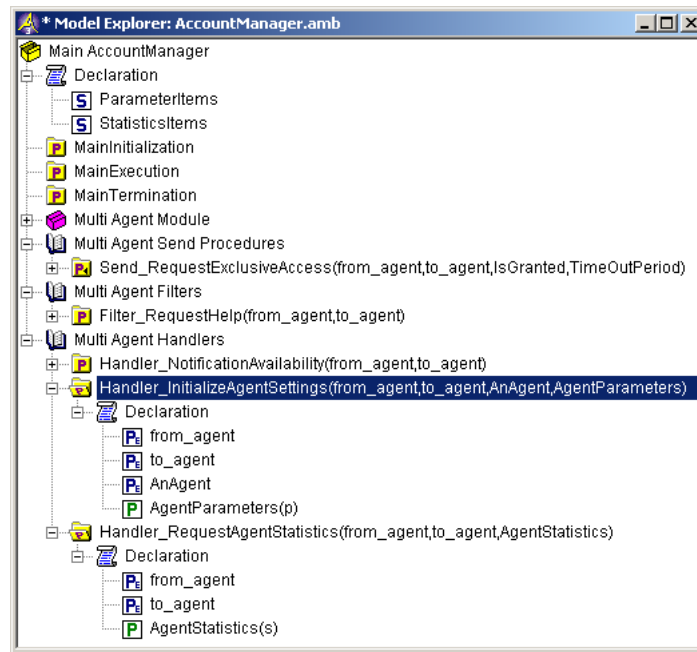


Figure 2.21: The handler procedures of the InitializeAgentSettings and RequestAgentStatistics messages

- DelayedFinishRequest,
- InitializeAgentSettings, and
- RequestAgentStatistics message types.

So far you have created the appropriate send procedures, handlers, and filters associated with all the message types in the community. In the next sections, we will continue with the specification of the procedure bodies of the handler and filter procedures for all message types.

Specifying the handler content

2.2.3 Handling initialization and statistics messages

Before starting the help desk simulation, the controller agent will send initialization messages to all agents. These messages contain the relevant settings for each agent role. Since these settings are required for the actual simulation, we will first implement the handlers for the initialization messages, and also add the necessary declarations for storing the settings in all agent projects.

Initialization messages

In order to store the agent settings supplied through the InitializeAgentSettings message, and the agent statistics requested through the RequestAgentStatistics message, you should add

Storing agent settings

- an element parameter MyAgent, and
- parameters MyParameters(p) and MyStatistics(s)

to the declaration section of the customer, account manager, and worker, agent projects as follows.

```
ELEMENT PARAMETER:
  identifier : MyAgent
  range     : MultiAgent::AllAgents ;
```

```
PARAMETER:
  identifier : MyParameters
  index domain : (p) ;
```

```
PARAMETER:
  identifier : MyStatistics
  index domain : (s) ;
```

For the account manager project this should result in a model tree as illustrated in Figure 2.22.

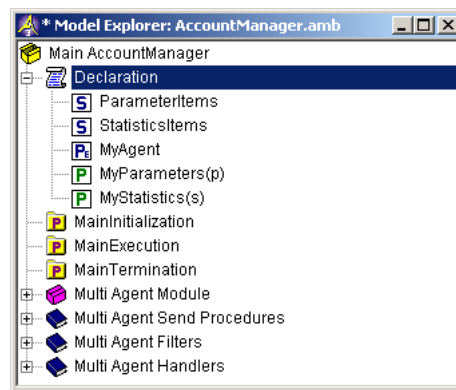


Figure 2.22: Global agent settings and statistics declarations

In the AIMMS code above—and throughout the remainder of this tutorial—we have chosen to always prefix the identifiers in the multi-agent module with “MultiAgent::”, as this gives you a clear indication when functionality from this module is used. However, since all identifiers in the agent module have been made public, you could also reference all identifiers without the prefix.

Protected versus public

Using these declarations, you can enter the following AIMMS code in the bodies of the Handler_InitializeAgentSettings procedures for the customer, account manager, and worker, projects.

Handling initialization messages

```
MyAgent      := AnAgent;
MyParameters(p) := AgentParameters(p);
```

The message handlers that take care of the actual help desk simulation will then use the relevant agent settings stored in the parameters `MyAgent` and `MyParameters(p)`.

Similarly, the `Handler_RequestAgentStatistics` procedure can be implemented for each of the customer, account manager, and worker, projects by inserting the following AIMMS code in their bodies.

Handling statistics requests

```
AgentStatistics(s) := MyStatistics(s);
```

The message handlers that take care of the actual help simulation will update the `MyStatistics(s)` parameter with statistics that are relevant for each agent role.

2.2.4 Generating, forwarding, and handling, help requests

This section contains the actual implementation of the help desk simulation. Since the whole implementation consists of loosely coupled message handlers, in various agent projects, responding to messages received asynchronously from other agents, you may find it hard to convince yourself that the provided implementation will actually perform the specified task. In this case, you are advised to re-examine the communication patterns shown in Figure 2.2 and the description of the message types in Section 2.1.1.

Simulating a help desk

We will start the implementation of the help desk application by implementing all the account manager related tasks. These are:

Implement account manager tasks

- dealing with incoming `RequestHelp` messages, and
- handling incoming `NotificationAvailability` messages.

Whenever a customer agent sends a `RequestHelp` message to its designated account manager, this message will be passed, by AIMMS' multi-agent layer, to the filter procedure `Filter_RequestHelp` in the account manager project. In this message filter, the account manager verifies whether a worker agent is available to deal with the request, and, if so, forwards the message to that worker. If no worker is available, the help request is placed in a queue and sent to a worker agent when one becomes available.

Handling help requests

To support the message filter for the `RequestHelp` message, account managers have to keep track of the availability of all workers. For this purpose, you can add

Auxiliary declarations

- a set declaration `AllWorkers` with index `w` and element parameter `AvailableWorker`, and
- a parameter declaration `IsAvailable(w)`

to the account manager project as described below. The definition of the set `AllWorkers` makes use of various identifiers in the multi-agent module to select all the agents with an agent role of 'Worker'. The element parameter `AvailableWorker` will be used to store the worker to which a help request will be forwarded.

```
SET:
  identifier   : AllWorkers
  subset of   : MultiAgent::AllAgents
  index       : w
  parameter   : AvailableWorker
  definition  : { MultiAgent::a | MultiAgent::AgentRole(MultiAgent::a) = 'Worker' };

PARAMETER:
  identifier   : IsAvailable
  index domain : (w)
  default     : 1 ;
```

We set the default of `IsAvailable(w)` to 1, so that we automatically pick up new worker agents joining the community.

Using the `IsAvailable(w)` identifier, and the `RequestExclusiveAccess` message type, an account manager can easily check for, and request exclusive access to, a worker marked as available, as implemented in the procedure `CheckForAvailableWorker` (without arguments) specified below. You should add this procedure to the account manager project, with the following AIMMS code as its body:

*The procedure
CheckForAvailableWorker*

```
empty AvailableWorker;

for ( w | IsAvailable(w) ) do
  Send_RequestExclusiveAccess( MultiAgent::ThisAgent, w, IsAvailable(w) );
  if ( IsAvailable(w) ) then
    AvailableWorker := w;
    break;
  endif;
endfor;
```

Note that the send procedure `Send_RequestExclusiveAccess` for the `RequestExclusiveAccess` message type is called in order to actually send the message to a worker `w`, updating its availability status `IsAvailable(w)`.

In its message handlers, each account manager agent keeps track of the following statistics in the identifier `MyStatistics(s)`:

*Account
manager
statistics*

- the number of forwarded messages (`MyStatistics('Forwarded')`), and
- the current queue size (`MyStatistics('Queued')`).

Using the procedure `CheckForAvailableWorker` and the message forwarding and saving functionality in AIMMS' multi-agent layer, the filter procedure `Filter_RequestHelp` can now be implemented in a straightforward manner. Enter the following AIMMS code as the body of the procedure `Filter_RequestHelp` of the account manager project:

*The procedure
Filter_
RequestHelp*

```

CheckForAvailableWorker;

if ( AvailableWorker ) then
  MyStatistics('Forwarded') += 1;

  MultiAgent::FilteredMessageForward(MultiAgent::ThisAgent,AvailableWorker);
  IsAvailable(AvailableWorker) := 0;
else
  MyStatistics('Queued') += 1;

  MultiAgent::FilteredMessageSave;
endif;

```

If the message is forwarded, AIMMS will directly relay the message to the specified worker agent. If it is saved, the message will be placed in a queue for later reference.

An account manager agent is notified when a specific worker becomes again available through a message of type `NotificationAvailability`. When account managers receive such a message, they will inspect their queue of unhandled `RequestHelp` messages. If their queue is non-empty, the account managers will try to claim the worker by sending a `RequestExclusiveAccess` message and, if successful, then forward the first unhandled `RequestHelp` message to that worker. By using the following functions

*Handling the
Notification-
Availability
message*

- `SavedMessageCount` to count the saved messages,
- `SavedMessageFind` to find the first unhandled message on the queue, and
- `SavedMessageForward` to forward an unhandled message

from the multi-agent module, the body of the message handler procedure `Handler_NotificationAvailability`, of the account manager project, can be implemented as below. Note that you need to add the declaration of a local parameter `FirstMessageOnQueue` to the procedure `Handler_NotificationAvailability` before compiling the procedure.

```

IsAvailable(from_agent) := 1;

if ( MultiAgent::SavedMessageCount ) then
  Send_RequestExclusiveAccess( MultiAgent::ThisAgent, from_agent,
                               IsAvailable(from_agent) );

  if ( IsAvailable(from_agent) ) then
    MyStatistics('Forwarded') += 1;
    MyStatistics('Queued') -= 1;

    FirstMessageOnQueue := MultiAgent::SavedMessageFind;
  end if;
end if;

```

```

MultiAgent::SavedMessageForward( FirstMessageOnQueue,
                                MultiAgent::ThisAgent, from_agent );
IsAvailable(from_agent) := 0;
endif;
endif;

```

This completes the implementation of all the account manager tasks for the help desk simulation. The model tree of your account manager project should look like the one illustrated in Figure 2.23.

The account manager model

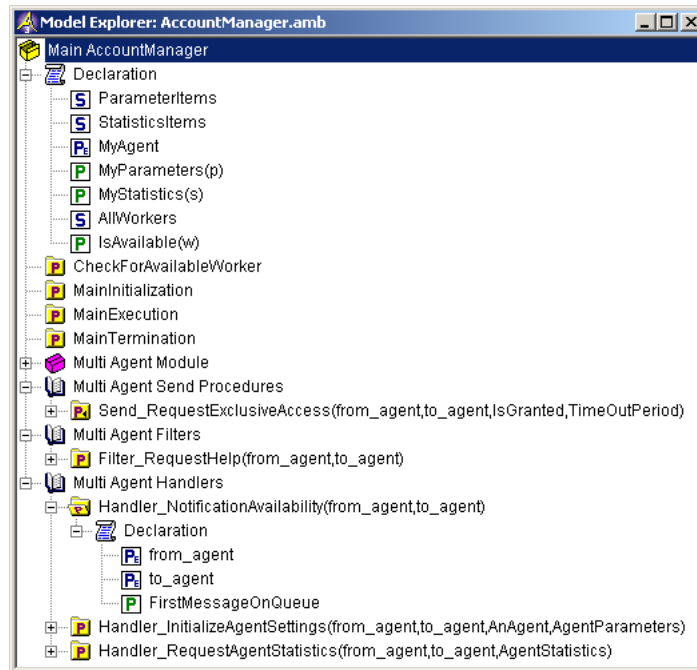


Figure 2.23: The model tree of the account manager so far

A worker agent only has two tasks, namely

- handling incoming RequestExclusiveAccess messages, and
- handling incoming RequestHelp messages.

Implementing worker tasks

Whenever worker agents receive a RequestExclusiveAccess message, they must verify whether they are available, and, if so, grant exclusive access to the claiming account manager.

Handling access requests

To support the handler for the `RequestExclusiveAccess` message, worker agents have to keep track of their availability status, of all available account manager agents, and should have access to the minimum and maximum work duration that were passed as arguments of the `InitializeAgentSettings` message, sent by the controller (see also Section 2.1.2). For this purpose, you should add

*Auxiliary
declarations*

- a parameter declaration `ThisAgentIsAvailable`, and
- a set declaration `AllAccountManagers`, with index `a`, and element parameter `ClaimingAccountManager`

to the worker project as described below. The definition of the set `AllAccountManagers` makes use of various identifiers in the multi-agent module to select all agents with the agent role of `'AccountManager'`. The element parameter `ClaimingAccountManager` will be used to store the account manager to which exclusive access has been granted.

```
PARAMETER:
  identifier : ThisAgentIsAvailable
  initial data : 1 ;

SET:
  identifier : AllAccountManagers
  subset of : MultiAgent::AllAgents
  index : a
  parameter : ClaimingAccountManager
  definition : { MultiAgent::a |
                MultiAgent::AgentRole(MultiAgent::a) = 'AccountManager' };
```

The parameter `ThisAgentIsAvailable` gets an initial value of 1 to indicate that each agent is available at startup. Note that you may get a compiler warning because AIMMS' multi-agent layer has not yet initialized the `AgentRole` parameter. You can safely ignore this warning.

When receiving a `RequestExclusiveAccess` message, the worker will agree to accept the work as long as he is not working *and* he has not been claimed by another account manager in the meanwhile. This process is captured in the following AIMMS code that should be entered as the body of the procedure `Handler_RequestExclusiveAccess` in the worker project.

*Handling the
RequestExclusiveAccess
message*

```
if ( ThisAgentIsAvailable and not ClaimingAccountManager ) then
  ClaimingAccountManager := from_agent;
  IsGranted := 1;
else
  IsGranted := 0;
endif;
```

Because there is no actual work to be done for a worker agent when handling a help request, we will split the help request into two sub-tasks to implement a random delay in a non-blocking manner (as discussed in Section 2.1.1):

Two sub-tasks

- handling the `HelpRequest` message, in which the worker agent sends a `DelayedFinishRequest` message to itself, and
- handling the `DelayedFinishRequest` message, in which the worker agent finalizes the help request and sends an `NotificationAvailability` message to all account managers.

Each worker agent uses the following agent parameters:

Worker settings

- minimum and maximum work duration (`MyParameters('Minimum duration')` and `MyParameters('Maximum duration')`).

Each worker agent keeps track, in its message handlers, of the following statistics:

Worker statistics

- the number of handled messages (`MyStatistics('Handled')`),
- the number of rejected help requests (`MyStatistics('Rejected')`), and
- the total duration of all handled requests (`MyStatistics('Duration')`).

To ensure that the account manager forwarding the help request is the same that has claimed the worker, the handler for the `RequestHelp` message will check the value of the element parameter `ClaimingAccountManager`, which had been set in the handler described earlier, against the account manager from whom the message has been received. This is implemented in the AIMMS code below, which you should insert as the body of the procedure `Handler_RequestHelp`. Note that you have to add the declaration of a local parameter `WorkDuration` to the procedure `Handler_RequestHelp` before compiling the procedure.

Handling the RequestHelp message

```

if ( from_agent = ClaimingAccountManager ) then
  ThisAgentIsAvailable := 0;
  empty ClaimingAccountManager;

  WorkDuration := Uniform( MyParameters('Minimum duration'),
                          MyParameters('Maximum duration') );
  MyStatistics('Duration') += WorkDuration;

  Send_DelayedFinishRequest( MultiAgent::ThisAgent, MultiAgent::ThisAgent,
                            SendDelay: WorkDuration );
else
  MyStatistics('Rejected') += 1;
endif;

```

A worker agent should only accept `DelayedFinishRequest` messages sent by itself, in which case `WorkDuration` seconds delay since sending the message will have passed. The worker agent can now finish handling the `RequestHelp` message by sending the `NotificationAvailability` message to all account manager agents. This is implemented in the following AIMMS code, which you should insert as the body of the procedure `Handler_DelayedFinishRequest`.

*Handling the
DelayedFinishRequest
message*

```
if ( from_agent = MultiAgent::ThisAgent ) then
  ThisAgentIsAvailable := 1;
  MyStatistics('Handled') += 1;

  for ( a ) do
    Send_NotificationAvailability( MultiAgent::ThisAgent,
                                  a ++ MyStatistics('Handled') );
  endfor;
endif;
```

Note that the above AIMMS code contains a trick in the second argument of the `Send_NotificationAvailability` call. By using the cyclic lead operator (`++`) in combination with `MyStatistics('Handled')`, one can, in a very simple manner, modify the order in which all account managers are notified of the availability of a worker agent. This prevents that the first account manager agent is always notified first, giving it an unfair opportunity to always claim a worker in advance of the other account managers.

This completes the implementation of all the worker tasks for the help desk simulation. The model tree of your worker project should look like the one in [Figure 2.24](#).

*The worker
model*

A customer agent has three tasks, namely

- handling incoming `StartSimulation` messages to start the simulation,
- generating `RequestHelp` messages at irregular points in time, and
- handling incoming `StopSimulation` messages to stop the simulation.

*Implementing
customer tasks*

The 'random' generation of `RequestHelp` messages is implemented in a non-blocking manner (see also [Section 2.1.1](#)) by sending a `DelayedGenerateRequest` message to the customer agent itself.

The customer project requires one additional global parameter, `SimulationInProgress`, as described below in order to indicate whether the simulation is currently in progress.

*Auxiliary
declarations*

```
PARAMETER:
  identifier: SimulationInProgress ;
```

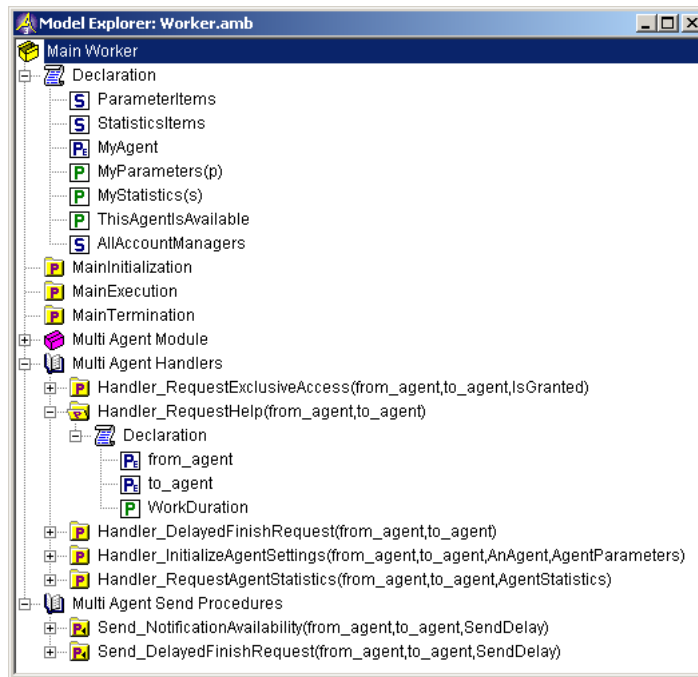


Figure 2.24: The model tree of the worker so far

Each customer agent uses the following agent parameters:

- its designated account manager agent (the MyAgent element parameter initialized when handling the InitializeAgentSettings message)
- the average rate at which requests are generated (MyParameters('Request rate')).

Customer parameters

Each customer agent keeps track in its message handlers of the following statistic:

- the number of generated help requests (MyStatistics('Generated')).

Customer statistics

A RequestHelp message is generated whenever the message handler of the DelayedGenerateRequest message is called. After sending the RequestHelp message to the customer's designated account manager, a new DelayedGenerateRequest message is sent to the customer agent itself with a random send delay. A customer agent will only handle DelayedGenerateRequest messages received from itself, and as long as the simulation remains in progress. This process is captured in the AIMMS code below that should be entered as the body of the procedure Handler_DelayedGenerateRequest in the customer project. Note that you need to add the declaration of a local parameter IntervalLength to the procedure Handler_DelayedGenerateRequest before compiling the procedure.

Random generation of RequestHelp messages

```

if ( from_agent = MultiAgent::ThisAgent and SimulationInProgress ) then
  MyStatistics('Generated') += 1;

  Send_RequestHelp( MultiAgent::ThisAgent, MyAgent );

  IntervalLength := Exponential( MyParameters('Request rate') );
  Send_DelayedGenerateRequest( MultiAgent::ThisAgent, MultiAgent::ThisAgent,
                               SendDelay: IntervalLength );
endif;

```

When a customer agent receives a `StartSimulation` message, the generation of `RequestHelp` messages has to be initiated. This can be done by sending an initial `DelayedGenerateRequest` message to itself. This is implemented in the AIMMS code below, which should be entered as the body of the handler procedure `Handler_StartSimulation` of the customer project. Note that you need to add the declaration of a local parameter `IntervalLength` to the procedure `Handler_StartSimulation` before compiling the procedure.

Starting the simulation

```

if ( not SimulationInProgress ) then
  SimulationInProgress := 1;

  IntervalLength := Exponential( MyParameters('Request rate') );
  Send_DelayedGenerateRequest( MultiAgent::ThisAgent, MultiAgent::ThisAgent,
                               SendDelay: IntervalLength );
endif;

```

When a customer agent receives a `StopSimulation` message, the generation of `RequestHelp` messages needs to be halted. This can be done simply by setting the value of the parameter `SimulationInProgress` to zero, which will cause the next call to the message handler of the `DelayedGenerateRequest` message to cease generating new `RequestHelp` messages. You should enter the following AIMMS code as the body of the handler procedure `Handler_StopSimulation` of the customer project.

Stopping the simulation

```
SimulationInProgress := 0;
```

This completes the implementation of all the customer agent tasks in the help desk simulation. The model tree of your customer project should look like the tree illustrated in Figure 2.25.

The worker model

2.2.5 Controller tasks

The controller agent has the following tasks:

Controller tasks

- starting the other agents in the community,
- initializing the other agents,

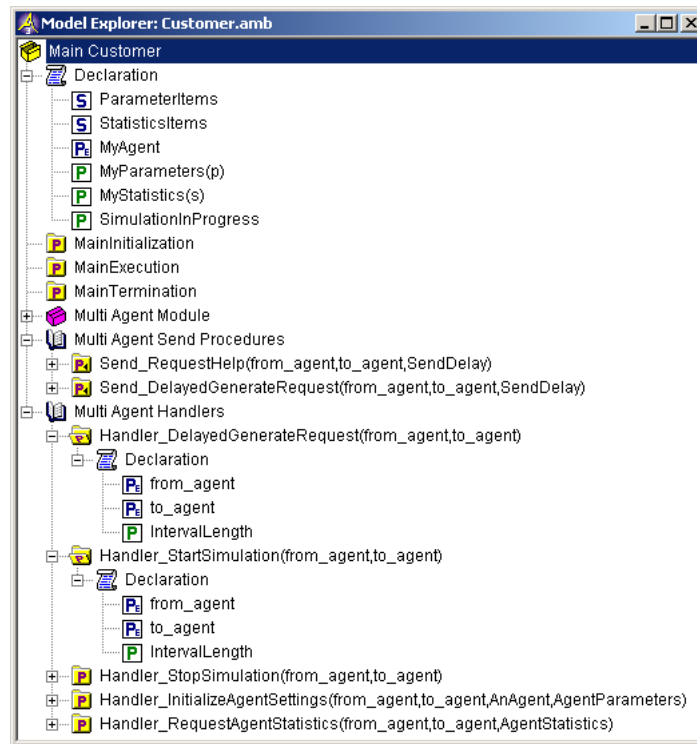


Figure 2.25: The model tree of the customer project so far

- starting and stopping the simulation, and
- monitoring other agents.

All tasks other than starting the other agents can be accomplished by using the messages implemented in the previous section. We will deal with these latter tasks first.

For initializing the other agents, the controller project contains a single procedure `InitializeOtherAgents`. In this procedure, the agent settings of all other agents will be determined and passed to the agents by sending `InitializeAgentSetting` messages.

Initializing the other agents

In support of the initialization of the agents, you need to add auxiliary declarations for the following identifiers to the controller project:

Auxiliary declarations

- the sets `AllCustomers` (with index `c`), `AllAccountManagers` (with index `a`) and `AllWorkers` (with index `w`),
- the element parameter `AllAgentAgent(MultiAgent::a)`, holding, for each agent, the `AnAgent` argument to be sent in the `InitializeAgentSettings` message, and

- the parameter `AllAgentParameters(MultiAgent::a,p)`, holding, for each agent, the `AgentParameters(p)` argument to be sent in the `Initialize-AgentSettings` message.

These declarations are listed below.

```

SET:
  identifier : AllCustomers
  subset of  : AllAgents
  index      : c
  definition : { MultiAgent::a | MultiAgent::AgentRole(MultiAgent::a) = 'Customer' } ;

SET:
  identifier : AllAccountManagers
  subset of  : AllAgents
  index      : a
  definition : { MultiAgent::a |
                MultiAgent::AgentRole(MultiAgent::a) = 'AccountManager' } ;

SET:
  identifier : AllWorkers
  subset of  : AllAgents
  index      : w
  definition : { MultiAgent::a | MultiAgent::AgentRole(MultiAgent::a) = 'Worker' } ;

ELEMENT PARAMETER:
  identifier : AllAgentAgent
  index domain : MultiAgent::a
  range       : AllAgents ;

PARAMETER:
  identifier : AllAgentParameters
  index domain : (MultiAgent::a,p) ;

```

Initially, the controller agent will generate random agent parameters for all customer and worker agents. This is implemented through the controller procedure `GenerateAgentParameters`, which you need to add to the controller project, and whose body is specified below. The procedure will also initialize the sets `ParameterItems` with all possible types of agent parameter types, and the set `StatisticsItems` with all possible types of agent statistics. Note that the `AnAgent` argument of the `InitializeAgentSettings` message is only used by customer agents to hold its designated account manager. We, therefore, only need to assign a value to the `AllAgentAgent` element parameter for customer agents.

*Generating
random agent
parameters*

```

ParameterItems := { 'Request rate', 'Minimum duration', 'Maximum duration' };
StatisticsItems := { 'Generated', 'Queued', 'Forwarded',
                    'Handled', 'Rejected', 'Duration' };

empty AllAgentAgent, AllAgentParameters;

! Initialize all customer agent settings
for (c) do
  AllAgentParameters(c,'Request rate') := Uniform( 0.6, 0.8 );

```

```

    AllAgentAgent(c) :=
        Element( AllAccountManagers, Mod( Ord(c) - 1, Card(AllAccountManagers) ) + 1 );
endfor;

! Initialize all worker agent settings
for (w) do
    AllAgentParameters(w,'Minimum duration') := Uniform( 2.5, 3.5 );
    AllAgentParameters(w,'Maximum duration') :=
        1.2 * AllAgentParameters(w,'Minimum duration');
endfor;

```

When all the agent parameters have been assigned appropriate values (either by calling the `GenerateAgentParameters` procedure declared above, or manually from within the end-user interface of the controller agent), all agents can be (re-)initialized using these agent parameters by calling the procedure `InitializeAllAgents` that you need to add to the controller project. Its body is specified below. Note that you need to add the declaration of a local set `InitializationMessages`, subset of the set `AllMessageTypes` (that has been declared in the multi-agent module), and the declaration of a local parameter `NumberOfMessagesHandled` before compiling the procedure.

*The procedure
InitializeAll-
Agents*

```

for ( MultiAgent::a | MultiAgent::a <> MultiAgent::ThisAgent ) do
    Send_InitializeAgentSettings( MultiAgent::ThisAgent, MultiAgent::a,
        AllAgentAgent(MultiAgent::a),
        AllAgentParameters(MultiAgent::a,p) );
endfor;

! wait for handled messages
InitializationMessages := {'InitializeAgentSettings'};

NumberOfMessagesHandled :=
    MultiAgent::WaitForMultipleHandledMessages( Card(AllAgents) - 1, AllAgents,
        InitializationMessages );

if ( NumberOfMessagesHandled <> Card(AllAgents) - 1 ) then
    DialogMessage( "Unable to initialize agent setting for all agents." );
endif;

```

The help desk simulation can now be started by sending the `StartSimulation` message to all customer agents. This is implemented in the procedure `StartSimulation` which you should add to the controller project. Its body is specified below.

*Starting the
simulation*

```

for (c) do
    Send_StartSimulation( MultiAgent::ThisAgent, c );
endfor;

```

Similarly, the simulation will be ended by send the StopSimulation message to all customer agents, as specified below in the body of the procedure StopSimulation that has to be added to the controller project.

Stopping the simulation

```
for (c) do
  Send_StopSimulation( MultiAgent::ThisAgent, c );
endfor;
```

Finally, we have to implement a procedure UpdateAgentStatistics to update the statistics provided by all the agents in the community. To store these statistics, you need to add an auxiliary parameter AllAgentStatistics(MultiAgent::a,s) to your model as listed below.

Requesting agent statistics

```
PARAMETER:
  identifier : AllAgentStatistics
  index domain : (MultiAgent::a,s) ;
```

This will hold the relevant statistics of all the agents in the community. With it, one can now implement the body of a procedure UpdateAgentStatistics as follows.

```
for ( MultiAgent::a | MultiAgent::a <> MultiAgent::ThisAgent ) do
  Send_RequestAgentStatistics( MultiAgent::ThisAgent, MultiAgent::a,
                              AllAgentStatistics(MultiAgent::a,s) );
endfor;
```

2.2.6 Remotely activating agents in the community

In this section we will discuss the process for remotely activating agent sessions. To activate another agent, the agent's AIMMS session needs to be first started, after which the agent needs to connect to the community.

Startup and agent creation

There are two ways in which an agent application can be started. An AIMMS application can be started manually, after which the connection to the community can be made. If an agent is started from within another agent, the connection has to be made immediately, i.e. the startup procedure has to contain instructions to connect to the community. Furthermore, if an agent is started remotely, the corresponding AIMMS application will open in end-user mode, thereby not allowing you to change the project. During development of your agents it is therefore recommended that you do not start up all agents remotely.

Manual vs. remote start

To remotely start (and stop) other agent applications, the following two procedures are available in the multi-agent module.

Remote start and stop

- StartRemoteProject
- StopRemoteProject

Note that the procedure StartRemoteProject will use the AIMMS COM object to start the new AIMMS session.

Every individual AIMMS project that is started up from within the controller agent must correspond to an element in the set MultiAgent::AllRemoteProjects. For such an element, the indexed string parameter MultiAgent::RemoteProjectPath should contain a reference to the path of the project that is to be started, and the indexed string parameter MultiAgent::RemoteProjectHost should contain the name of the host computer on which the project is to be started. If the project host is not specified, the agent will be started on the local host, and the path specified in RemoteProjectPath may contain a relative path. In this case, the new AIMMS session will be started in a minimized state. If a host is specified (including "localhost"), the project path *must* be an absolute path, and the new AIMMS session on that host will be started as a server application (i.e. not visible).

The set AllRemoteProjects

The following declarations, which you should add to the declaration section of the controller project, are used in the procedures StartRemoteProjects and StopRemoteProjects discussed below. Note that you may get a compiler error about non-existing agent roles 'Customer', 'AccountManager' and 'Worker', because the multi-agent layer has not yet initialized the set AllAgentRoles. You can ignore this error, and save the declaration NumberOfAgent(ar) without compiling it.

Auxiliary declarations

```
PARAMETER:
  identifier   : NumberOfAgents
  index domain : (MultiAgent::ar)
  initial data : data { Customer : 4, AccountManager : 2, Worker : 8 }
```

```
PARAMETER:
  identifier   : ProjectStarted
  index domain : (MultiAgent::arp) ;
```

The procedure StartRemoteProjects, which you should add to the controller project and which is implemented below, automatically creates elements in the set AllRemoteProjects. The procedure also creates the corresponding entries for the string parameter RemoteProjectPath, pertaining to every customer, account manager and worker that plays a role in the example. Note that you should add the declarations of a local string parameter NewRemoteProjectPath, and of a local element parameter NewRemoteProject into the set AllRemoteProjects to the procedure StartRemoteProjects before compiling it.

Starting remote projects

```

for (MultiAgent::ar) do
  switch (MultiAgent::ar) do
    'AccountManager' :
      NewRemoteProjectPath := "..\\AccountManager\\AccountManager.prj";
    'Customer'       :
      NewRemoteProjectPath := "..\\Customer\\Customer.prj";
    'Worker'         :
      NewRemoteProjectPath := "..\\Worker\\Worker.prj";
  endswitch;

  while ( Loopcount <= NumberOfAgents(MultiAgent::ar) ) do
    SetElementAdd( MultiAgent::AllRemoteProjects, NewRemoteProject, FormatString(
      "%e-%n", MultiAgent::ar, Loopcount ) );
    if ( not ProjectStarted( NewRemoteProject ) ) then
      MultiAgent::RemoteProjectPath( NewRemoteProject ) := NewRemoteProjectPath;
      MultiAgent::StartRemoteProject( NewRemoteProject );

      ProjectStarted( NewRemoteProject ) := 1;
    endif;
  endwhile;
endfor;

```

Note that this implementation allows you to increase the number of agents to be started for a particular agent role, and then re-run the procedure to start these new agents as well. Furthermore, all agent sessions will be started visibly and in minimized state, because no host is specified. By specifying a host, and providing a absolute project path, the agents can be started invisibly.

The procedure `StopRemoteProjects`, implemented through the AIMMS code below, stops all remote projects started earlier through a call to `StartRemoteProjects`.

Stopping remote projects

```

for (MultiAgent::arp | ProjectStarted(MultiAgent::arp)) do
  StopRemoteProject(MultiAgent::arp);

  ProjectStarted(MultiAgent::arp) := 0;
endfor;

```

After starting an AIMMS agent session (whether manually or remotely), that agent session must request the multi-agent layer of AIMMS to create a queue for other agents to communicate with. When creating such a queue, AIMMS initially assumes that this queue forms the beginning of a new agent community. However, by connecting this new agent queue with another (known) agent queue, the queues will be joined to form one larger community. Note that AIMMS only allows you to join an empty queue with an existing community. Finally, the newly created agent session must register all of the agents it hosts (along with their roles) with that community. Therefore, the process of connecting an AIMMS agent session to an existing community can be partitioned into three phases:

Forming an agent community

1. creation of the queue used by the agent,

2. joining the community associated with the queue of one of the existing agents, and
3. registering one or more agents with the community.

The controller agent in this example is special in the sense that it is the agent that is started manually, as the first agent of a new agent community to be formed. All other agents become part of the the controller's community by connecting to its queue. Before starting up the other agents, the controller agent must therefore run a procedure `StartCommunity` which

- creates a queue with a known name (i.e., "Controller queue"), and
- registers itself with the corresponding community as the controller agent.

The body of the procedure `StartCommunity` is specified below. You should add this procedure to the controller project.

```
MultiAgent::CreateLocalQueue( "Controller Queue" );
MultiAgent::Logon( agentname : "Controller" );
```

Since the queue of the controller agent has a known queue name, the remaining agent sessions started by the controller agent can connect to this queue, and, thus, enlarge the community started by the controller agent. The following AIMMS code takes care of this process, and you should add this as the body of a `AgentStartup` procedure to be included in the customer, account manager, and worker, projects. Note that we are assuming that all agents run on the same computer. If the remote queue resides on another computer the procedure `ConnectToRemoteQueue` should be called using a second optional argument indicating the name of the computer that hosts the remote queue.

Start up of the other agents...

```
MultiAgent::CreateLocalQueue;
MultiAgent::ConnectToRemoteQueue( "Controller Queue" );
MultiAgent::Logon;
```

When an agent session is started remotely, from within another agent in the community, this new agent is expected to join the community and log on its corresponding agents before the end of the call to the `StartRemoteProject` function. To accomplish this, you should make the `AgentStartup` procedure, as just implemented, the startup procedure of the customer, the account manager, and the worker, agent project, as illustrated in the **Options** dialog box in Figure 2.26.

... must be called during creation

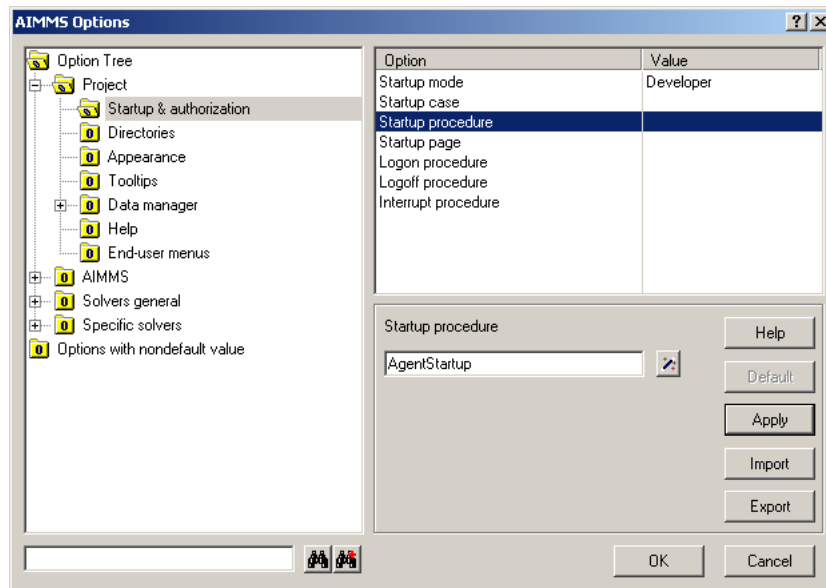


Figure 2.26: The AIMMS Options dialog box showing the startup procedure

2.2.7 A controller user interface

In this tutorial, the controller agent will be the agent that starts all the others and gathers the statistics. Using all the parameters and procedures created in the previous section, it is not difficult to create a user interface for the controller agent. With such a user interface, one can easily start and stop the other agents, and monitor the behavior of the overall community. Figure 2.27 shows an example of an overview page that can be created for the controller agent.

The user interface

2.3 Running the agent application

Although all aspects of all four agent roles have now been discussed, it might be a good idea to finish reading this section before actually running the application. This section contains several useful tips (e.g. debugging tips) that could be of help when running the application.

This section

2.3.1 Verifying community behavior

A first step in verifying community behavior is to check whether all agents are present in the community. Since, for every agent, a single AIMMS session

Checking whether all agents are alive

The screenshot shows a window titled "Controller page" with two tables and a control panel on the right.

AllAgentParameters Table:

	AllAgentParameters			AllAgent.Agent
	Request rate	Minimum duration	Maximum duration	
Controller				
Customer_1	0.77			AccountManager_1
Customer_2	0.61			AccountManager_2
Customer_3	0.65			AccountManager_1
Customer_4	0.68			AccountManager_2
AccountManager_1				
AccountManager_2				
Worker_1		3.19	3.83	
Worker_2		3.32	3.98	
Worker_3		3.00	3.60	
Worker_4		2.60	3.12	
Worker_5		3.27	3.92	
Worker_6		2.59	3.10	
Worker_7		2.77	3.32	
Worker_8		3.27	3.92	

AllAgentStatistics Table:

	AllAgentStatistics					
	Generated	Queued	Forwarded	Handled	Rejected	Duration
Controller						
Customer_1	7					
Customer_2	5					
Customer_3	5					
Customer_4	6					
AccountManager_1			12			
AccountManager_2		1	10			
Worker_1				2		10
Worker_2				2		11
Worker_3				2		10
Worker_4				2		8
Worker_5				1		7
Worker_6				2		8
Worker_7				2		9
Worker_8				1		7

Control Panel Buttons: Start community, Start remote agents, Stop remote agents, Generate agent parameters, (Re-)initialize agents, Start simulation, Stop simulation, Update statistics.

Figure 2.27: The user interface used in the controller agent

should have been started by the controller, your task bar should contain AIMMS sessions for all the agents. Furthermore, every agent should correspond to an element in the set `AllAgents` that is declared in the multi-agent module. This set should contain all agents that make up the community. The contents of this set is displayed in the upper left table of the page shown in Figure 2.27.

By inspecting the statistics it should become clear that all customers are indeed generating help requests, that all account managers are forwarding these requests to one of the workers, and that all workers indeed do contribute to the handling of the help requests.

Should the statistics indicate some problem with one of the agents, the cause might be the retrieval of the statistics themselves. To rule this possibility out, it is best to inspect the agent itself. For this purpose, it is a good habit to also equip the other agents with a startup page that displays some relevant agent data (as is the case in the example project).

The behavior of the system is naturally also influenced by the number of agents and their corresponding parameters. Parameters that influence the balance between the generation and handling of requests are

- the request rate of the customers, and
- the average work rate of the workers.

Interpreting controller statistics

Inspecting statistics of other agents

Tuning input data

If you have completed the page illustrated in Figure 2.27, you can run the simulation as follows

Running the application

- ▶ press the **Start community** button, this will set up the controller queue,
- ▶ press the **Start remote agents** button, this will start up all the remote agents and log them onto the community,
- ▶ press the **Generate agent parameters** and **(Re-)initialize agents** buttons, this will compute the agent parameters and initialize the agents with them, and
- ▶ press the **Start simulation** button to start the actual simulation.

Once the simulation is underway, you can regularly inspect the agent statistics by pressing the **Update statistics** button. You will see that, with the default agent settings, the workers will not be able to keep up with the generated requests.

By pressing the **Stop simulation** button, the customer agents will be instructed to stop generating help requests, allowing the worker agents catch up with the queued requests. If all agents have finished their tasks, you can press the **Stop remote agents** button to close all AIMMS sessions of the customer, the account manager, and the worker agents.

Stopping the simulation

2.3.2 Debugging a multi-agent application

Investigating the cause of communication problems between two agents may be a difficult task. To help you locate the problem, you can use

Debugging multi-agent applications

- the AIMMS debugger, or
- the log generated by AIMMS' multi-agent layer.

Using the AIMMS debugger requires that the AIMMS session at hand be started in developer mode. Therefore, if you want to use the AIMMS debugger to debug a multi-agent session, such an AIMMS session must be started manually, as sessions started by a call to `StartRemoteProject` are always started in end-user mode. Within the debugger, you can set breakpoints in message filters and message handlers and examine the message arguments. Note, however, that use of the debugger in message handlers with output arguments, or of response messages, may result in a time out for the sending agents who are awaiting a response message.

The AIMMS debugger

It is possible to log all message related execution to the AIMMS **Message Window**. To enable message logging, you should set the option `External_message_filter` to 'All'. This option can be found in the **AIMMS - External function** category of the **Options** dialog box. It will cause the **Message Window** to display an overview of all message communication from and to one specific agent.

Message logging

A global overview of the message flow within the community is not available using this option. Figure 2.28 shows the **Message Window** for the controller project after initializing the agent parameters, starting the simulation and updating the agent statistics.

```

Messages
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Customer_1
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Customer_2
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Customer_3
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Customer_4
(1030624445) Send: message:InitializeAgentSettings from:Controller to:AccountManager_1
(1030624445) Send: message:InitializeAgentSettings from:Controller to:AccountManager_2
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Worker_1
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Worker_2
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Worker_3
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Worker_4
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Worker_5
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Worker_6
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Worker_7
(1030624445) Send: message:InitializeAgentSettings from:Controller to:Worker_8
Waiting ... (timeout: 10000 milliseconds)
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:AccountManager_1 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Customer_3 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Customer_1 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Customer_4 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Customer_2 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:AccountManager_2 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Worker_1 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Worker_2 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Worker_3 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Worker_4 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Worker_5 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Worker_6 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Worker_7 to:Controller
(1030624445) ReceivedResponse: message:InitializeAgentSettings from:Worker_8 to:Controller
Waited successfully for 14 messages
(1030624468) Send: message:StartSimulation from:Controller to:Customer_1
(1030624468) Send: message:StartSimulation from:Controller to:Customer_2
(1030624468) Send: message:StartSimulation from:Controller to:Customer_3
(1030624468) Send: message:StartSimulation from:Controller to:Customer_4
(1030624482) Send: message:RequestAgentStatistics from:Controller to:Customer_1
Waiting ... (timeout: 5000 milliseconds)
(1030624482) ReceivedResponse: message:RequestAgentStatistics from:Customer_1 to:Controller
Waited successfully for 1 message
(1030624482) Send: message:RequestAgentStatistics from:Controller to:Customer_2
Waiting ... (timeout: 5000 milliseconds)
(1030624482) ReceivedResponse: message:RequestAgentStatistics from:Customer_2 to:Controller
Waited successfully for 1 message
(1030624482) Send: message:RequestAgentStatistics from:Controller to:Customer_3
Waiting ... (timeout: 5000 milliseconds)
(1030624482) ReceivedResponse: message:RequestAgentStatistics from:Customer_3 to:Controller
Waited successfully for 1 message

```

Figure 2.28: The **Message Window** of the controller project

Although the actual agent application to be developed may consist of many agents, running on multiple computers, it is good practice to start with a small number of agents, all residing on a single computer, while debugging. This prevents you from getting lost in flood of, possibly bogus, messages, and allows you to quickly kill agents that lock up manually using the Windows task manager. For example, in this tutorial, to test whether the agents behave as expected you might as well set up a community that contains only one customer, one account manager and one worker.

Debug a small community on a single computer

2.3.3 Miscellaneous tips and tricks

When starting a new project, AIMMS will, by default, create a `MainTermination` procedure that asks whether unsaved data must be saved, when the project is closed. This feature may be convenient in an interactive application, but can be rather annoying when remotely closing other agents in the community. You prevent this behavior by simply changing the body of the `MainTermination` procedure to

```
return 1;
```

*Change
MainTermination*

When several agents are running, it may be difficult to find out which AIMMS instances correspond to which agents. To distinguish between individual agents, you can modify the project title of the main AIMMS window, for instance, to reflect the agent type, or, even, the individual agent name. The project title can be set through the `Project_title` option that can be found in the **project - Appearance** category of the **Options** tree. You can also set this option dynamically through the AIMMS function `OptionSetString`.

*Use project title
to distinguish
agents*