# User's Guide and Reference Manual for the Agent Library provided as part of SPADES Version 0.7

Patrick Riley `<pfr+@cs.cmu.edu>`

March 27, 2003

## 1 Introduction

This is the primary documentation for the agent library provided as part of the System for Parallel Agent Discrete Event Simulation (SPADES). This document does not cover the functioning of the main SPADES system, which is covered in other documentation available at the above URL.

The agent library provides a C++ interface to the interaction of an agent with the communication server. The library provides basic support for sending and receiving messages in the correct format. The agent library uses a variety of classes from the main SPADES library, and provides four primary additional classes (and some subclasses):

`AgentCommInterface` This class in the main point of interface. All messages are sent and received through this class.

`ToCSMessage` This is a virtual base class. Subclasses provide all the messages that can be sent to the communication server.

`FromCSMessage` This is a virtual base class. Subclasses provide all the messages that can be received from the communication server.

`AgentLibraryParam` This class is a subclass of `ParamReader` (described in the main SPADES documentation) which consists of the parameters for controlling the agent library.

The agent library also provides the action and error logging facilities discussed in the main SPADES documentation.

## 2 `AgentCommInterface`

This class is the main point of interface with the agent library. It provides functionality to send and receive messages.

The methods are:

**Constructor** Sets up for reading and writing to the file descriptors specified by the parameters *input_fd* and *output_fd*, but does not send any messages.

**Destructor** Closes the pipes for communication with the communication server.

`tryMessageRead` Tries to read a message from the communication server and return it. This method never blocks and returns `NULL` if no message is available.

`receiveMessage` Like `tryMessageRead`, except it blocks waiting for a message. Will call `select` waiting for a message at most *num_select_timeouts* times, each time waiting the value specified by *wait_sec* and *wait_usec*. Returns `NULL` (and sets the shutdown flag) if there is a timeout or an error on the receiving pipe.

`shouldShutdown` Returns whether the shutdown flag is set, either by a call to initiateShutdown, an error reading from the input pipe, or by a timeout.

`initiateShutdown` Sets the shutdown flag to true.

`getLatestTime` Returns the latest time stamp on any message received so far.

`send` Sends a message (of type `ToCSMessage` to the communication server.

`getOutstandingThinks` Gets the number of outstanding think messages. Unless there is an error, this will always be 0 or 1. In other words, if the agent is in the middle of thinking cycle, this will be 1, and 0 otherwise.

## 3 `ToCSMessage`

This is a virtual base class for messages that can be sent to the communication server from the agent. The methods on the base class are:

`Print` This prints a text based representation of the message for use with the `<<` operator. It is intended primarily for debugging and informational purposes.

`getOutstandingThinkMod` Returns how this message affects the number of outstanding thinks (done thinking returns −1, sensations return +1, etc.).

`writeTo` This writes the message in the format that the SPADES communication server understands.

There is one subclass of `ToCSMessage` for each message that the agent can send to the communication server. Several of these classes use `DataArray` which is provided by SPADES and described in the main documentation.

`ToCSMessage_Act` Takes a `DataArray` as an argument for the act message to send.

`ToCSMessage_RequestTimeNotify` Takes a integer for the simulation time for which to request the time notify.

`ToCSMessage_MigrationData` Takes a `DataArray` with the data to pass to the new instance of this agent.

`ToCSMessage_DoneThinking` The done thinking messages tells the communication server that this agent has finished the thinking cycle.

`ToCSMessage_Exit` Notifies the communication server that the agent is exiting.

`ToCSMessage_InitDone` Notifies the communication server that initialization has completed.

## 4 FromCSMessage

This is a virtual base class for messages from the communication server to the agent. The methods on the base class are:

`Print` This prints a text based representation of the message for use with the `<<` operator. It is intended primarily for debugging and informational purposes.

`getOutstandingThinkMod` Returns how this message affects the number of outstanding thinks (done thinking returns −1, sensations return +1, etc.).

`accept` In order to avoid the user of this library form having to use run time type casts or other painful things, the class here follow the Visitor design pattern. In order to handle each different message specially, you can subclass from `FromCSMessage::Visitor` or `FromCSMessage::ConstVisitor`. There is one visit method on each of those classes for each subclass of `FromCSMessage`. When you get a message to process, you can call the `accept` method with an object which has type of your subclass of `FromCSMessage::Visitor` or `FromCSMessage::ConstVisitor`. The appropriate `visit` method of your object will be called.

There is one subclass of `FromCSMessage` for each message that the agent can receive from the communication server. Several of these classes use the `DataArray` which is provided by SPADES and described in that documentation. The subclasses of `FromCSMessage` are:

`FromCSMessage_InitData` After every startup, and initialization data message is sent. If this is not migration then the data will be empty.

`FromCSMessage_Exit` Tells the agent to exit. No more messages will be sent or received.

`FromCSMessage_Sense` This is a sensation. It may or may not start a thinking cycle (use the `getThinking` method to find out). You can access the data and times associated with the sensation.

`FromCSMessage_TimeNotify` The time notify that is sent for every request time notify message. It may or may not start a thinking cycle (use the `getThinking` method to find out).

`FromCSMessage_MigrationRequest` Tells the agent to send back a migration data message since it is about to be migrated.

`FromCSMessage_Error` An error message from the communication server. The error token is stored as a string. The possible values and their meanings can be found in the main SPADES documentation.

`FromCSMessage_ThinkTime` Notifies the agent of the amount of thinking time used for its last thinking cycle.

## 5 Parameters

The class `AgentLibraryParam` contains the following parameters. It further provides facility for parsing parameters on the command line or in files, as described for the `ParamReader` class in the main SPADES documentation.

You will probably want to subclass `AgentLibraryParam` in order to add your own parameters.

| Name | Type<br>Description | Default |
|---|---|---|
| *logfile_dir* | String<br>The directory in which to put logfiles. | Logfiles |
| *action_log_fn* | String<br>The name of the file for the action log (recording debugging information). See the main SPADES documentation for details. | actions.log |
| *action_log_level* | Integer<br>The highest action log level to write to the action log file. See the main SPADES documentation for details. | 0 |
| *random_seed* | Integer<br>The number to seed the random number generator with. Any negative value causes a new seed to be read from `/dev/urandom`. | -1 |
| *wait_sec* | Integer($\geq 0$)<br>The seconds to wait in each `select` call inside of `receiveMessage`. | 5 |
| *wait_usec* | Integer($\geq 0$)<br>The microseconds to wait in each `select` call inside of `receiveMessage`. | 0 |
| *num_selects_timeout* | Integer($\geq 0$)<br>The number of `select` calls without a message being received before the agent exits. | 20 |
| *input_fd* | Integer($\geq 3$)<br>The file descriptor on which to read messages from the communication server. This should match the value given in the agent type database. | 3 |
| *output_fd* | Integer($\geq 3$)<br>The file descriptor on which to send messages to the communication server. This should match the value given in the agent type database. | 4 |

## 6 Example of Use

The sample agent provides a more complete example of how to use the agent library. The section provides some psuedo-code describing the steps you probably want to go through.

```
Allocate a parameter object (subclass of AgentLibraryParam)
Call getOptions on that object
Logger::instance()->setTagFunction(new AgentTagFunction);
Logger::instance()->setMaxActionLogLevel(
    AgentLibraryParam::instance()->getActionLogLevel());
spades::seedRandom(AgentLibraryParam::instance()->getRandomSeed());
Allocate pacsi, of type AgentCommInterface
FromCSMessage* m;
while ( (m = pacsi->receiveMessage()) != NULL )
    Process message m
    delete m;
```