MACSimJX: A Tool for Enabling Agent Modelling with Simulink Using JADE

Charles R. Robinson, Peter Mendham, and Tim Clarke

***Abstract*—MACSimJX provides the means for advanced modelling and development of multiagent driven control systems. This is achieved by drawing together two modelling tools used extensively in their respective communities. These are Simulink, a tool used for control systems development and JADE, an environment for developing agents. Thus the strengths of their particular domains of application may be drawn upon to facilitate research and development in the joint field of decentralised systems control. To the authors knowledge no other implementation such as this exists. MACSimJX, otherwise known as the extension of MACSim with JADE, is available for download at www.agentcontrol.co.uk.**

***Index Terms*—MACSimJX for Decentralised Control, Simulink with JADE, Agent-Based Systems, Sensor Fusion, Control Architectures and Programming.**

I. Introduction

In general, a decentralised system has its processing distributed such that each element in the system is capable of functioning in isolation. However, there is the potential for enhanced system performance because these processing units can communicate and cooperate with each other. Decentralised systems offer a number of advantages over their centralised counterparts that includes greater robustness, timeliness and fault tolerance. This paper reports an integrated software framework that connects control system simulation with multiagent theory to support the modelling of real-time decentralised systems.

Multi-agent architecture, a concept that began to be properly developed in the latter half of the 1980s [1], provides a natural software support structure for decentralised systems. In this context the word *agent* is used to refer to a software entity capable of operating by itself, with the ability to obtain information from, and effect changes on, its environment. These operations may include communication with other agents and are carried out in order to achieve some predefined objectives. There are many potential advances in system design that might be achieved through the development and application of the emerging multi-agent technology.

Simulink is a widely used tool in industry and academia. It is a graphical front-end for MATLAB (Matrix Laboratory) which allows representation of time varying systems through matrix manipulation. Simulink provides a graphical representation of these systems modelled through matrices. A vast array of libraries are available that provide the ability to connect a series of subsystem elements to construct and represent the internal mechanics of such dynamic and embedded systems.

Charles R. Robinson is with THALES R&T France.

Peter Mendham is with SciSys UK Ltd.

Tim Clarke is with The University of York.

MACSimJX provides access from Simulink to such a multiagent architecture, this facilitates the development of software control structures with features such as:

- Robustness, so that if part of the system fails the rest of the system will continue to operate with minimal loss of functionality.
- Scalability of the processing architecture.
- Fewer constraints on a system caused by computational bottlenecks or communication bandwidth.
- Modularity in terms of both the design and implementation.
- Synergy of sensors such that the overall machine perception is improved beyond basic fusion of data.
- An enhanced awareness of the state of the world.

Thus MACSimJX is an interface that enables models of systems created in Simulink to exchange data with a multiagent system created using JADE. A brief description of the agent paradigm and JADE follows, after which the manner in which MACSimJX integrates this with Simulink is discussed.

II. Intelligence and Multiple Agents

The word *agent* refers to a concept rather than to a particular entity or event. Concepts are general ideas that describe a class or category of things or events that have unique features but share common characteristics. Like other concept words such as tree or dinosaur the meaning of the word agent encompasses a set of ideas that people believe represent its general properties. This means that although it is possible to describe agents and trees, many traits or characteristics will depend on circumstances, the environment and our own experience. With this important qualification, a general definition for an agent is:

***Definition*:** An agent is an autonomous entity in an embedded environment that either solves problems by itself, or cooperates with other agents to find a solution. It has control over its internal state as well as its outputs and can run without external intervention.

An embedded environment implies a system that receives its information about the environment through sensors and acts on the environment through effectors. A multi-agent system is a collection of these interacting agents and can exhibit all the features required in a decentralised setup. It has the capacity to achieve tasks through the combined efforts of the individual agents that could not be done alone. This is particularly the

case where the agents have different functional capacities. Agent systems have also been shown to exhibit emergent problem-solving behaviours through cooperation not explicitly part of the original design, one such example being swarm intelligence.

Jennings *et al.* [2] list many of the applications to which agents have been applied in industry, commerce, medicine and the entertainment sector. However, these authors stated that the agent community suffered from the lack of a systematic design methodology or an industrial strength multi-agent toolkit. Since then, these shortcomings have been addressed, or are being addressed, by FIPA and the development of several agent frameworks. A software agent framework is one that incorporates the various agent theories and provides a support structure of and for functions that facilitate the rapid development of a system based on these concepts.

JADE provides a framework that allows quick implementation of many of the inherent features one would expect for developing a multi-agent system. Much of the complexity is kept hidden from the user to allow ease of use. In addition, JADE has been steadily gaining support and has been more widely used over the last few years. For these reasons, and with support from a review in [3], JADE was chosen to be the framework to assist agent modelling for Simulink through MACSim.

A. The Java Agent Development Environment (JADE)

JADE was originally developed under TILab, formerly CSELT, in Italy [4] to address the lack of support available for building agent systems. As its name suggests, this framework offers an environment in which to create agents written in Java. It provides the runtime environment, which the agents require in order to operate. It also provides an extensive library of classes with methods built around the FIPA specification of agent characteristics and graphical interfaces for monitoring active agents.

Each instance of a runtime environment is called a *container* and several of these make up a *platform*. The first container to be created needs to be designated as the *main container*; subsequent containers then register with this as they join the platform. Containers can be spread across several networked computers. The main container hosts an agent management service (AMS) and a Directory Facilitator (DF). The AMS ensures that each agent has a unique name and can be used for loading and removing agents from the platform. The DF provides a means for agents to publicise their specialised services or for looking up the services provided by other agents. It is often referred to as the *Yellow Pages* [5].

Agents operate from within the containers. The structure of an agent consists of a *setup()* method, one or more behaviour methods, and a *takeDown()* method. The setup() method is executed the first time an agent is created and runs only once. It sets all the initial conditions needed to get the agent up and running and includes the behaviours required for the agent.

The behaviour methods can run concurrently and are responsible for carrying out the main tasks of an agent. This includes communicating with other agents. An agent can be put to *sleep* if it has no behaviours operating and can be awakened again after a specified period, or on receipt of a message requesting the execution of an action. This can be very useful because the agent consumes no processing power when it is in sleep mode.

To date, JADE has been applied, at least in theory, to a wide variety of areas including urban and aircraft traffic control [6], [7], providing travel industry support [8], manufacturing [9] and robotics [10], [11].

III. Architecture

Whilst Simulink is really effective for carrying out simulations, it falls short of offering the tools necessary to set up an agent framework. One very useful aspect of Simulink, however, is that it provides a work-around for adding functionality in the form of S-functions. These allow programs to be written in other languages, particularly C, that can be encapsulated in the Simulink environment and then used where desired, running in their native language.

Despite this prospect of a solution, where the agents could be created through C++ or Java code in one of these functions and run in Simulink, there is a further complication. S-functions are unable to handle multiple threads of execution: they become unstable if several processes run concurrently inside Simulink [12], [13]. Unfortunately, this functional property is essential for a multi-agent system. To overcome this problem, a program called MACSim was created which still utilises the S-function ability of Simulink, but only as a gateway to pass data to a program outside MATLAB with parallel processing capacity.

A. Structure of MACSim

MACSim, or the *Multi-Agent Control for Simulink* program, described in [12], was purposely developed as a medium through which a program for implementing agent designs developed in C/C++ or Java might pass data to and from Simulink. Although MACSim is written primarily in C++, it includes a wrapper to enable interaction with Java programs. MACSim has a client-server architecture, where the client part is embedded in Simulink through an S-function, and the server code is then incorporated in the separate program as indicated in Figure 1. Communication between the client and server is then performed through the use of *named pipes* in Windows. Use of MACSim circumvents the multi-threading issue because a separate program can now be used with protocols in place to ensure synchronicity if so desired.

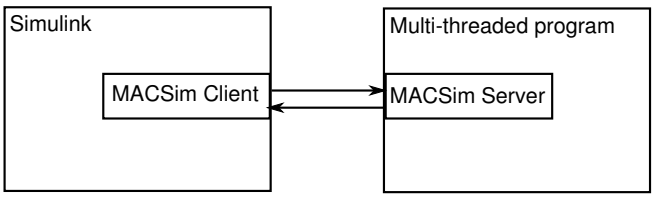

Fig. 1. Structure of MACSim.

While a developer has the option of writing their own C++ or Java agents from scratch, it is frequently more efficient to build on what is already available. Thus MACSimJX extends the functionality of the server side of MACSim to allow Simulink to interact specifically with JADE. The developer thus has the added capabilities of this agent development environment at their disposal. This extension using JADE is described next.

B. Extending MACSim to use JADE

MACSimJX provides the means, utilising JADE, to receive data from Simulink via the MACSim interface and to pass this on to relevant agents for processing. Once the agents have finished working on the data, the data must be returned to Simulink along the same channels. The agents are designed to accomplish some goal, such as optimisation of incoming data.

For this purpose it seemed logical to divide the agent model into two parts, the Agent Environment (AE) and the Agent Task Force (ATF). The environment is a transformation of the MACSim server, previously mentioned, to provide a transparent connection between Simulink and the JADE agents. It contains the ground-work required for any generic agent model spread across Simulink and a JADE program, including responsibility for passing any data between the two programs.

The other part, the ATF, contains the agents responsible for interacting with the Simulink data. Some simple protocols need to be followed by the agents of the ATF to ensure the appropriate exchange of data with the AE. For all other purposes, these agents may be developed as normal, with the behaviours and goals one may wish to see implemented in a real-life system. The arrows in Figure 2 outline the communication paths for the three sections of the complete model.

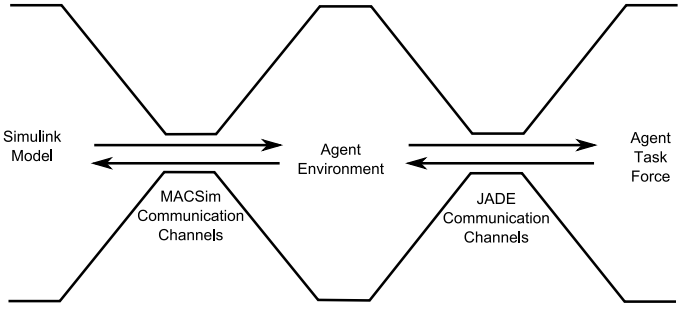

Fig. 2. Outline of the complete model.

The rest of this section considers the JADE classes for the agent environment and the basic communication standards that need to be built into agents designed for the ATF. It focuses on the characteristics of the code and classes provided by the framework to get an agent up and running. JADE has a framework where a good proportion of the underlying structure, used for implementing agents, is deliberately hidden from the user to avoid making the development cycle over-complicated.

Someone using JADE will have functions they wish their agents to perform and, at least initially, will only want to concern themselves with the programming of these functions. It should be possible to place these straight into some agent template in order to get their agents up and running. JADE provides what is termed an API (Application Programming Interface) which is effectively a library describing the different classes and functions with the parameters they require and return after running. These classes provide the backbone for agent development with JADE, at least in terms of the general agent properties, and make the whole process relatively painless. In the API it is possible to search for the properties one wishes an agent to exhibit, including those related to agent behaviours, communication methods between agents, and the resulting interaction of agents.

To assist with this rapid prototyping of agents, a template is suggested. Derived from [14] and [4], the skeletal code for this is shown in Listing 1. It can be divided into several major parts. The agent code commences by importing the various libraries of functions that include those required for use in the agent being designed. The class name of the agent type being created is then declared. Inside this are the two main functions, *setup()* and *takeDown()*, their names being sufficient to describe their purpose. Following these is a selection of inner classes, in this case one, containing the various behaviours the agent will exhibit and which are to be initialised through the setup() method. There are some basic behaviour types provided by JADE, such as the OneShotBehaviour (executes once) and the CyclicBehaviour (repeats its code continuously). These behaviours can be utilised to create customised behaviours in which the programmer places the desired agent functions and also the code for communication with other agents.

Thus, the code of Listing 1 can be filled out fairly easily, with the desired agent properties expressed inside the behaviour functions. Implementing the agent is then simply a case of executing the JADE *runtime environment* from command prompt and calling the relevant agents. Detailed examples are provided in the JADE tutorial guide [14], Programmer's Guide [4] and Administrators Guide [15].

C. The Agent Environment

The AE acts as an interface for the JADE agents and Simulink. It has been suggested [12] that such an interface should be responsible for the following:

- Keeping track of all current agents and facilitating *the dynamic 'birth' and 'death' of agents.*
- Synchronisation with Simulink through MACSim.
- Providing the current input and time step data when requested.
- Storage of data to be output back to Simulink and allowing for these data to be altered.
- Having the capability to broadcast messages to the agent population.

The first requirement indicated above is handled automatically by JADE through its DF which is, in effect, an agent that acts like the 'yellow pages' where agents register with the services they can offer and can search for those they require. Synchronisation is optional, agents can either wait for data

Code listing 1

```
import jade.lang.acl.*;
import jade.core.Agent;
import jade.core.behaviours.*;

public class SkeleAgent extends Agent {
// Initialise class variables.

  protected void setup() {
/* Attempt to initialise agent,
 * including its various behaviours.
 * Add the behaviour for receiving
 * agent messages.
 */

  addBehaviour(new AgentBehaviour1());
  Object[] args = getArguments();
  if (args != null && args.length > 0) {
  // Operate on the received parameters
  // provided by agent initiator as args.
  }
  else {  // Unable to create agent.
    System.out.println(No arguments.);
    System.out.println(Ending agent.);
    takeDown();
    }
  }

  protected void takeDown() {
  // Remove agent from system.
  }

  class AgentBehaviour1
    extends CyclicBehaviour {
    public void action() {
      ACLMessage msg = receive();
      if (msg != null) {
      // Process the message
      }
    }
  }
}
```

from Simulink before carrying out any actions, or also perform tasks while waiting for new data. The other requirements mentioned above are handled by the various components of the AE that will now be described.

The code for the agent environment is composed of seven classes, two of which represent actual agents. The core classes are the *AgentServer*, *AgentCoordinator*, *EnvironmentAttributes* and the *TimeStepData* class. The other three provide some extra functionality for designing agents and are the *TimeProfiler*, *UsefulAgentMethods* and the *Matrix* class. The methods contained by these classes are detailed in the MACSimJX API. The AE is outlined in Figure 3. Here you see a more detailed representation of the Agent Environment with the two JADE agent classes, AgentServer being responsible for interacting with Simulink and AgentCoordinator managing exchanges with the ATF.

The afore mentioned responsibility of current input and time step is handled by the EnvironmentAttributes class, along with the number of inputs to and from Simulink (these can be accessed by the relevant *get* and *set* methods). Storage of data is done in the TimeStepData class which keeps track of changes made by the agents until it is ready to be sent back to Simulink. Finally through the JADE services functionality, any agent subscribed to the ”Agent” service will receive any broadcast to this service.

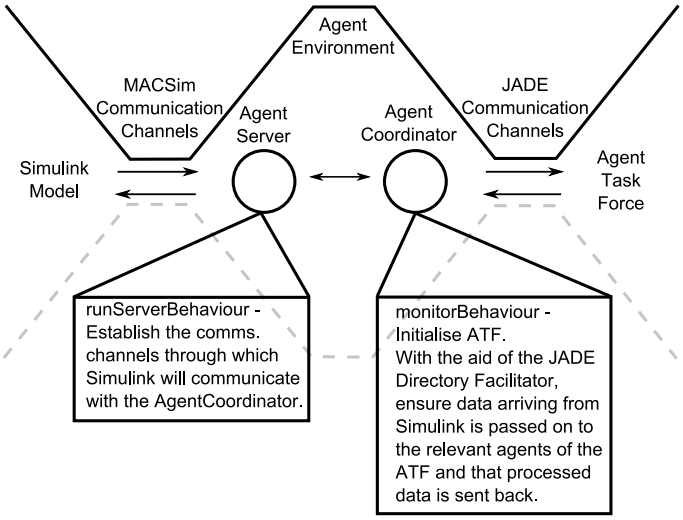

Fig. 3. An outline of the Agent Environment.

D. The Agent Task Force

The ATF consists of all the agents that jointly operate on the data arriving from Simulink in order to accomplish some task. All of these agents will have *setup()* and *takedown()* functions and probably some behaviour functions. However, with the exception of communication protocols with the AE, the implementation of the functions will take on very different forms depending on the particular task the designer wishes them to achieve.

An example is shown shortly to demonstrate the implementation of an ATF. However, there are several common features it is appropriate to draw to the reader’s attention beforehand. An overview is provided in Figure 4, where between the *First Agent* and *Last Agent*, there can be any number of other agents. Each having the ability to communicate with the other agents of the task force, and with the AgentCoordinator from the AE. The diagram depicts the general behaviour of the *Last Agent*. As an example, it is given a filter behaviour, but with the exception of applying the Kalman filtering, the same series of steps apply to the other agents.

Each designed ATF is ideally given its own *sub-package* with inner packages for each type of agent it uses. The agent

package contains the agent class along with any associated classes to supplement the agent class. Inspection of the ATF diagram shows that there are also three standard messages that a designer may wish to implement in their agents. These messages being identified as:

- UpdateData - Provides the new data arriving from Simulink at each sample step.
- DataAmended - A confirmation that data amended by an agent has been received by the AE.
- Shutting Down - An instruction received from another agent to end current processes and terminate.

These three messages are primarily for interaction between the agents of the AE and ATF. It is assumed the developer would wish the freedom to determine the mechanisms of communication within their own ATF. Some of these agents may have no need to communicate with the AE, relying on others to pass on the needed data.

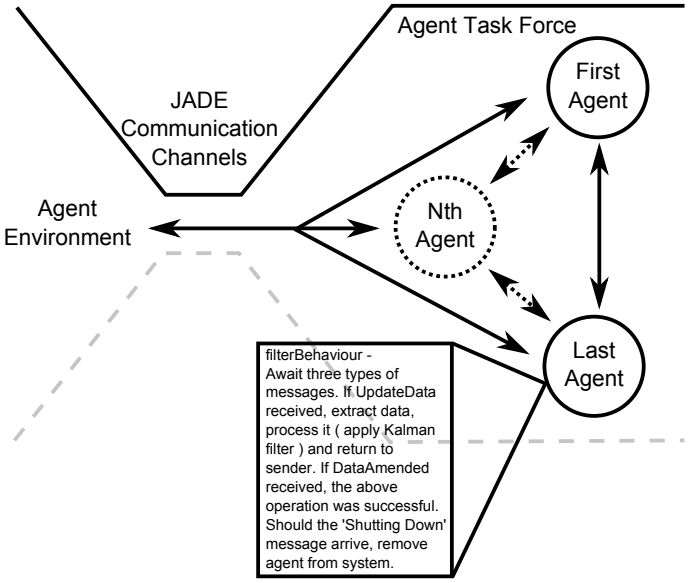

Fig. 4. Overview of the ATF.

Simulink model performs the same arithmetic operations as used by the agents. This is to provide a comparison with the agent output for the sake of validation.

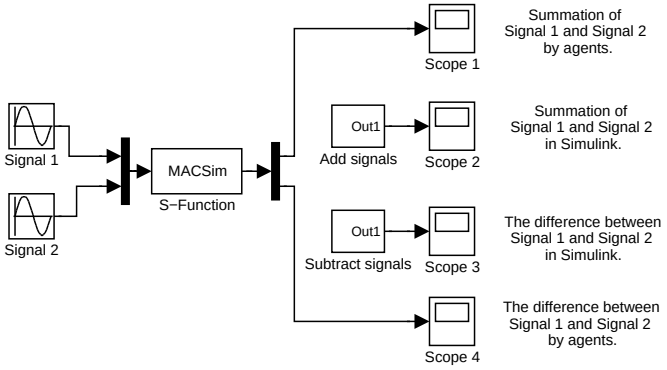

Fig. 5. Simulink example model.

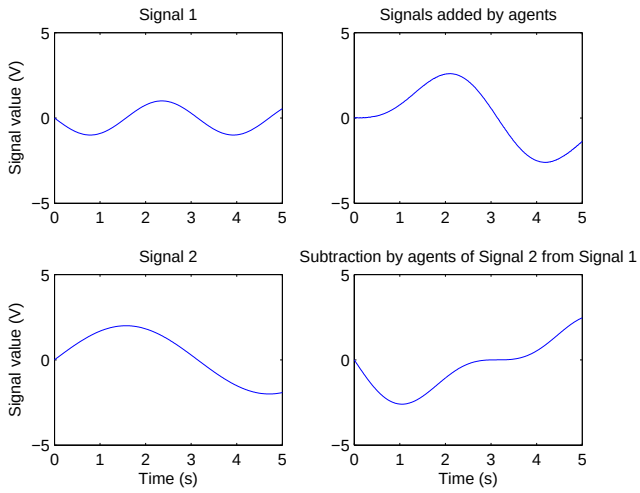

Fig. 6. Scope output.

IV. Demonstration

This section provides an example of the procedure that is followed in order to set up some agents for an ATF. For the sake of clarity a straightforward scenario is used. Two different sinusoidal-type signals are fed from Simulink through MACSimJX. Both signals are then communicated to the ATF. In this example, the ATF consists of two agents. These will apply some arithmetic to the signals, that is, one agent finds the sum of the signals and the other finds the difference, and these results are sent back to Simulink.

The initial step is to set up the model being used in Simulink. This is shown in Figure 5, where the two signal generation blocks are shown connected to the MACSim block. The MACSim block is the client that will be exchanging data with the agents. The outputs from the MACSim block are then connected to scopes for analysis. The final component of the

The next stage is to develop the agents required for the ATF. This commences by customising the JADE agent template, shown in Listing 1, for use with MACSimJX, and then incorporating the arithmetic operations into the agent behaviours.

In order for agents to interact with MACSimJX, there are three message types the agent should be prepared to receive. These have the ID tags of *UpdateData*, *DataAmended* and *Shutting Down*, as mentioned in the previous section. The most important of these messages, for agent operation, is UpdateData which consists of a data structure containing an array representing the input ports of the MACSim block, and the current data sample at each of these ports. Upon receiving such a message, the agents carry out their processing, in this case some arithmetic, and then return a message to the sender, usually the AE, containing a data structure with the results and indicating the elements that were changed. To illustrate this part of the design, with an agent interacting with MACSimJX,

the behaviour segment of Listing 1 is extended and shown in Listing 2.

Following the design of the agents, all that is left is to run this ATF alongside Simulink. To do this, the agent class files should be placed in an appropriately named folder under MACSimJX\ATFs. MACSimJX is then executed (by running a .jar file), which opens a GUI allowing the relevant details to be entered, such as adding the location of the agents class files, the number of inputs\outputs of the MACSim block in Simulink and the sample rate. With these details completed, the *Continue* button is clicked and our agents are now ready to operate. Finally, the Simulink model is opened, the simulation time specified and then set to run. Figure 6 shows sample data that has been run through the system for five seconds.

The TimeProfiler class provides some information about the ratio of time spent in Simulink and the AE to the time in the ATF. In the current example, due to the simplicity of the simulation, about 99 percent of the time was spent on the JADE side. TimeProfiler additionally provides methods to locate bottlenecks that might exist in a developers agent code in order to assist with this aspect.

A more advanced example of using MACSimJX may be found in [3], where a Boeing 747 is modelled in Simulink and data from its sensors are sent to agents for fusion where centralised and decentralised Kalman filters are tested.

V. Conclusion

An overview has been provided of a useful enabling tool called MACSimJX. The desirable nature of decentralised multi-agent-driven systems were discussed. Simulink, an industry standard program for modelling dynamic real-time systems, was introduced and its shortcomings with respect to multi-threading were described. MACSimJX provides a bridge to rectify this problem, the support taking the form of JADE.

Thus an integrated software framework is available for the development, testing and analysis of multi-agent control systems. It incorporates an interface (MACSimJX) that enables the co-simulation of dynamic systems (under Simulink) and a multi-agent system (JADE). This opens up a very wide field of investigation, in particular for the design of complex and dynamic multi-agent control systems.

Abbreviations

AE	Agent Environment.
AMS	Agent Management Service.
API	Application Programming Interface.
ATF	Agent Task Force.
DF	Directory Facilitator.
FIPA	The Foundation for Intelligent Physical Agents.
JADE	Java Agent Development Environment.

Code listing 2

```
class AgentBehaviour1
  extends CyclicBehaviour {
  public void action() {

/* Initialise data structure,
 * and other local variables.
 */
    // Prepare agent to receive a message.
    ACLMessage msg = receive();

    if (msg != null) {
    String message=msg.getConversationId();

      if (message.equals("UpdateData")) {
      //In this example, from the AE.

/* Try to extract data structure from
 * the new message, and from this,
 * the the data array and length.
 */
/* The agent then performs any data
 * operations required before other
 * ATF agents are considered.   If these
 * initial results are required by other
 * ATF agents, they are sent as a data
 * structure in a  new message.  Otherwise,
 * the agent waits until receiving all
 * expected data from other ATF agents.
 */
/* For this example the agent now finds
 * either the sum or the difference
 * between the two data array elements.
 * Having completed the calculations,
 * results are returned to the AE.
 */
      }
      if (message.equals("DataAmended")) {
      // i.e. a response from AE confirming
      // receipt of data from this agent.

      // If no further operations required,
      // end current conversation with AE.
        replyToAgent(msg.getSender(),
          "ProcessingComplete");
      }
      if (message.equals("Shutting Down")) {
        // Terminate the agent.
        takeDown(msg.getSender());
      }
    } else {
      // Put agent to sleep.
      block();
    }
  }
}
```

REFERENCES

[1] M. Wooldridge and N. R. Jennings, "Intelligent agents: theory and practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[2] N. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998.

[3] C. R. Robinson, "Decentralised data fusion using agents," Ph.D. dissertation, The University of York, 2008.

[4] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa, *Jade Programmers Guide*, TILab S.p.A., 2005.

[5] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "Jade - a white paper, in "exp in search of innovation - special issue on jade," TILAB, Tech. Rep., 2003.

[6] Z. Li, F.-Y. Wang, Q. Miao, and F. He, "An urban traffic control system based on mobile multi-agents," in *IEEE International Conference on Vehicular Electronics and Safety*, Beijing, China, October 2006, pp. 103–108.

[7] M. Pěchouček, D. Šišlák, D. Pavlíček, and M. Uller, "Autonomous agents for air-traffic deconfliction," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, Hakodate, Japan, May 2006, pp. 1498 – 1505.

[8] B. Balachandran and M. Enkhsaikhan, "Development of a multi-agent system for travel industry support," in *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce*, Sydney, Australia, December 2006, p. 63.

[9] D. Naso and B. Turchiano, "A coordination strategy for distributed multi-agent manufacturing systems," *International Journal of Production Research*, vol. 42, pp. 2497–2520, 2004.

[10] J. Eze, H. Ghenniwa, and W. Shen, "Distributed control architecture for collaborative physical robot agents," in *IEEE International Conference on Systems, Man and Cybernetics*, Washington, D.C., USA, October 2003, pp. 2977 – 2982.

[11] P. Santana, V. Santos, and J. Barata, "Dsaar: A distributed software architecture for autonomous robots," in *IEEE Conference on Emerging Technologies and Factory Automation*, Prague, Czech Republic, September 2006, pp. 1017–1020.

[12] P. Mendham and T. Clarke, "Macsim: A Simulink Enabled Environment for Multi-Agent System Simulation," in *Proceedings of the 16th IFAC World Congress, Prague, Czech Republic*, 2005.

[13] J. M. Solanki and N. N. Schulz, "Using intelligent multi-agent systems for shipboard power systems reconfiguration," in *Proceedings of the 13th International Conference on Intelligent Systems Application to Power Systems*, November 2005.

[14] G. Caire, *Jade Programming for Beginners*, TILab S.p.A., 2003.

[15] F. Bellifemine, G. C. amd Tiziana Trucco, G. Rimassa, and R. Mungenast, *Jade Administrator's Guide*, JADE Board, 2006.