

# DECAF Programming: Agents for Undergraduates<sup>\*</sup> † ‡

Foster McGeary  
Computer and Information Sciences  
University of Delaware  
mcgeary@cis.udel.edu

Keith Decker  
Computer and Information Sciences  
University of Delaware  
decker@cis.udel.edu

## ABSTRACT

Agent programming has required mastery of several concepts before working agents could be developed. Those concepts include programming in a high-level language, understanding communication protocols, “thinking distributedly,” and project management. With DECAF, we make advances in reducing the level of proficiency required with these concepts to develop useful working agents. We believe undergraduates can reasonably be expected to build working agents with DECAF. Specification and explanation of DECAF appears in other work. This paper is intended to assist the user unexperienced with DECAF to “get up and running” with simple agents.

## 1. DISTRIBUTED ENVIRONMENT CENTERED AGENT FRAMEWORK – DECAF

DECAF is a toolkit which allows a well-defined software engineering approach to building multi-agent systems. The toolkit provides a stable platform to design, rapidly develop, and operate intelligent agents to achieve solutions in complex software systems. DECAF provides the necessary architectural services of a large-grained intelligent agent: communication, planning, scheduling, and execution monitoring. This is the internal “operating system” of a software agent, to which application programmers have limited access.

Control and programming of DECAF agents is assisted by the graphical “PlanEditor” user interface. In the PlanEditor, executable actions are basic building blocks that can be chained together to achieve large, complex goals in the

<sup>\*</sup>Based on work supported by the National Science Foundation under Grant Nos. IIS-9733004 and IIS-9812764.

<sup>†</sup>A full version of this paper is available as *DECAF Programming: An Introduction* at [www.cis.udel.edu/~decaf](http://www.cis.udel.edu/~decaf)

<sup>‡</sup>DECAF itself, along with the working demonstration, is available at the same website.

style of a hierarchical task network. This approach provides a software component-style programming interface with desirable software properties including component reuse (eventually, automated via the planner) and some design-time error-checking. The chaining of activities can involve traditional looping and if-then-else constructs. This part of DECAF is an extension of the RETSINA and TAEMS task structure frameworks.

### 1.1 Advanced DECAF Agent Programming

Unlike traditional software engineering, each action can also have attached to it a performance profile which is then used and updated internally by DECAF to provide real-time local scheduling services. The reuse of common agent behaviors is thus increased because the execution of these behaviors does not depend only on the specific construction of the task network but also on the dynamic environment in which the agent is operating. For example, a particular agent is allowed to search until a result is achieved in one application instance, while the same agent executing the same behavior will use whatever result is available after a certain time in another application instance. This construction also allows for a certain level of non-determinism in the use of the agent action building blocks. This part of DECAF is based on the Design-to-Time/Design-to-Criteria scheduling work at the University of Massachusetts [4]. We do not address these advanced uses in this paper.

### 1.2 DECAF Architecture

Figure 1 shows the structure of DECAF. Concern in this paper is with the boxes outside the large block labeled “DECAF Task and Control Structure.” We will discuss Plan Files, KQML messages (both outgoing and incoming) and the Action Modules (Java code). Domain Facts and Beliefs are the facts and beliefs that the agents users have about the domains in which they are solving problems with DECAF. Reference [1, 2] describe the insides of the box.

### 1.3 Experience the Demonstration

DECAF has a demonstration accessible through the DECAF homepage. The demonstration produces a five-agent world:

- **ANS** is the Agent Name Service, which is *essential* to have running for any DECAF agent application to find the addresses of other agents (“white pages”).
- **ANSQuery** is an optional tool that lists the names that ANS has assigned to agents that requested them.

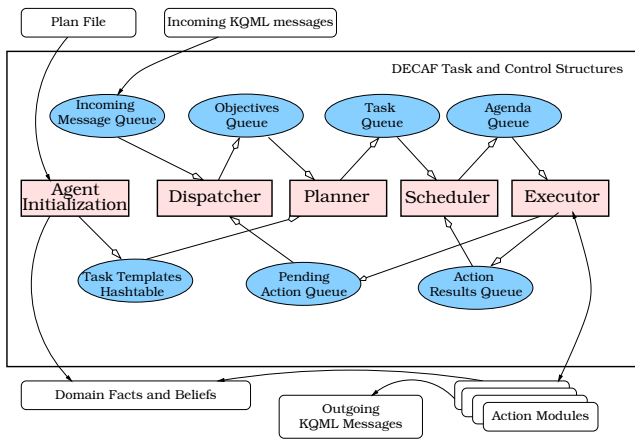


Figure 1: DECAF Architecture Overview

- **Matchmaker** is optional, but the demonstration uses it. Matchmaker accepts “advertisements” from agents and provides the names of advertising agents to any agents that request such names.
- **SDBW** is a Simple Data Base Wrapper that advertises with Matchmaker (“yellow pages”). It advertises itself as a provider of services, and Matchmaker stores the keywords by which it would like to be found. (See [3] for a discussion of the Matchmaker.)
- **SIA** is a Simple Interface Agent. The demonstration has you use the SIA to retrieve an entry from the SDBW. The SIA uses the Matchmaker to locate all services that offer information using certain keywords, lets the user pick one to interrogate (the demonstration has only one, SDBW), and asks that agent for a match to the user’s query.

The remainder of this paper assumes the reader has downloaded and worked the demonstration. Only by doing so can the reader most profitably continue with this paper.

## 2. SPECIFY DECAF AGENT BEHAVIOR

There are several related steps in specifying the behavior of DECAF agents, which steps are together referred to as the programming of a DECAF agent. Fortunately, only two pieces of software are involved in that programming. The two pieces of software are the DECAF PlanEditor and Java. We will discuss the DECAF PlanEditor in the next subsection and then we will show some agent Java code.

### 2.1 The PlanEditor

The PlanEditor is a graphical interface used to create the programming for the Agent. PlanEditor is the software that draws the lines and boxes and clouds that specify to DECAF what an agent is enabled to do. (That is, what *could* be done, provided DECAF has access to the right Java code to execute and has been sent the right messages to initiate it.) Start the PlanEditor while in the SDBW directory of the demo. Clicking on the standard “file” then “open” menu items will show the collection of “.lsp” files from which to select a plan file to edit. There is only one that comes with the demonstration and is in this directory, namely “SDBW.lsp,” which, when highlighted and selected shows a graphical “picture” of the agent.

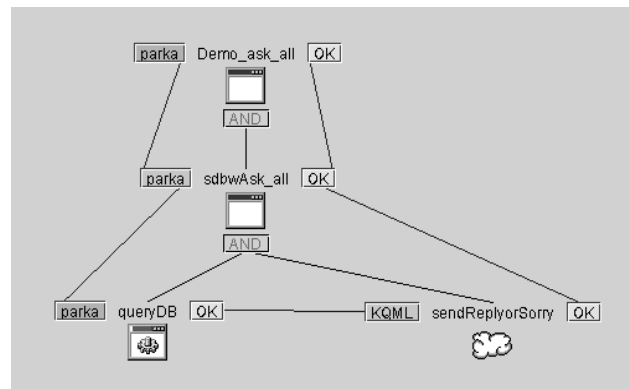


Figure 2: The Demo\_ask\_all Task Structure

#### 2.1.1 Plans Look Like Forests of Trees

The Program is a tree-like structure (Hierarchical Task Network, or “HTN”). The demonstration has a simple database wrapper agent (“SDBW”) with three such tree-like structures:

- **Demo\_ask\_all** which is the Task for the SDBW to answer queries.
- **\_Startup** which initiates the SDBW agent, including registering with the Matchmaker.
- **\_Shutdown** which closes down the SDBW agent, including unregistering with the Matchmaker.

Graphically, there are no connections between the three structures. There is no need to connect them because they are separate entities (“Tasks”) within the agent. These three Tasks all belong to the same agent because they are in the same plan (.lsp) file. Since DECAF is multi-threaded, DECAF could have all tasks in a plan hierarchy executing at the same time, but usually will not. In the present case, the logic of the agent strongly suggests that these three tasks will *not* be executing at the same time. One Task Structure, “Demo\_ask\_all” is shown as Figure 2.

#### 2.1.2 Components of a Task Structure

Each tree represents one task structure and also usually represents one Java code file, revealing that there is close to a one-to-one correspondence of DECAF Tasks and Java code files. Each leaf nodes is an action, and each must have a like-named method in the associated Task Java class. Each node (leaf and non-leaf) has 0 or more inputs (provisions or parameters) and 1 or more outcomes. Figure 2 shows all the major features of a DECAF agent Task:

- **Top-level Tasks** are the highest level task abstracts, and may or not correspond to Java classes. Since there is no code for “Demo\_ask\_all” available to DECAF, it passes control to the next lower task. Note also that the name of a Top-level Task must be the concatenation of an ontology and a task name.
- **Intermediate Tasks** are tasks invoked by other DECAF tasks. The lowest level task is embodied in Java code as a Java class with the same name as the name given to the Task icon. For Figure 2, there must be a Java file named sdbwAsk\_all.java that has been compiled into the Java class named sdbwAsk\_all.class for the application to work. Observe that DECAF will

reason about the Characteristic Accumulation Function of a Task, but that it will not do so for Actions. (See Section 2.2.3.)

- **Actions** are Java methods that DECAF invokes for the agent. The way to distinguish between Tasks and Actions is that Actions have a little gear in their icons, while Tasks have none. Actions are embodied in Java code as a method with the same name as the name given to the Action icon that is found in the class with the same name as the name given to the Task icon.
- **Non-local Tasks** are represented as clouds, which represent the mechanism by which messages are sent by this agent to an arbitrary agent (that is, either itself or some other DECAF agent) to have that other agent (hence, “non-local”) perform some task. That task is always a high-level Task at the receiving agent.

The following conventions apply to agent Tasks, Actions, and Clouds (non-local Tasks):

- Inputs to an icon (called “provisions” or “parameters”) are the items on the left of the icon.
- Icon output is one of the outcomes on the icon’s right.
- No two icons can have the same name.
- Each icon except the top-level icon is owned by a higher level icon.
- Actions and clouds cannot own other icons.
- Icons have their ownership indicated by lines from their owning task to the top of the owned icon.
- The destination for the message string associated with an output is indicated by a line drawn from the output side of an icon to either
  - the left-hand side of another icon, indicating that this output from the source icon is the input to the destination icon, or
  - the output node of another icon, indicating that this output from the source icon is to be treated as the output of the destination icon. This only works for Actions that provide outputs for Tasks.

Naming of the items needs to be careful on two accounts. First, there can be no duplicate names in the plan, so each must be unique, and the PlanEditor will enforce this. Second, having names that are clear without being overly long makes matching plans with Java code easier. Naming clouds is best achieved by merging the agent(s) sent the message with the intended result, such as “gopher\_selectsSeat” to indicate that the “gopher” class of agent will select a seat for the user. As always, naming is hard to do well.

Names for provisions are fairly easy – the application will suggest them. Names for outcomes will need to match the corresponding field in the Java code, and it is **very important** that the names and values in the code and in the plan are consistent, properly spelled, and, where possible, meaningful.

### 2.1.3 Programming the Agent - Briefly

Here is a skeleton of the Java code in the sdbwAsk\_all.java file that would be compiled into the sdbwAsk\_all.class needed for the above example to work. We will revisit this code in a subsequent section. The actual source code for sdbwAsk\_all.java comes with the demonstration.

```
public class sdbwAsk_all
{
    public sdbwAsk_all()
    { }
    public ProvisionCell queryDB(LinkedListQ Plist, Agent Local)
    { KQMLmsg K = new KQMLmsg();
      return new ProvisionCell(K.getKQMLString(),"OK");
    }
}
```

Note the following in and about the Java code:

- The class name is the same as the lowest level **Task** name in the plan.
- There must be a null constructor for the class.
- There is a method for each action that the Task owns.
- Method names conform **exactly and precisely** to the Action names in the plan file.
- There is no option to the programmer in what can be passed to the methods. DECAF always provides a LinkedListQ and an Agent.
- Each method must return a ProvisionCell.
- Provision cells consist of two strings, in this case a KQMLmsg converted to a string and the string “OK”.
- The contents of the second string in the ProvisionCell returned from the queryDB method must match exactly the outcome box in the plan for the queryDB action (see Figure 2).

Not spelling names of methods correctly prevents DECAF from finding the code to execute it. Not coding the returns from methods properly either gives DECAF an undocumented outcome, which it will not be able to deliver anywhere (thus jamming the thread), or it causes the method to look to DECAF as if it had left the action through one outcome when the user intended another (thus making the agent “act funny”).

## 2.2 Particular Details of Plan Creation

The PlanEditor is a straightforward GUI, and experimenting with it is encouraged. The major functions to accomplish with it are discussed briefly below. We discuss creating new items, the modification of items (both new and existing), selecting a behavior for the task relative to its actions, and how to draw the lines.

### 2.2.1 Creating a New Item

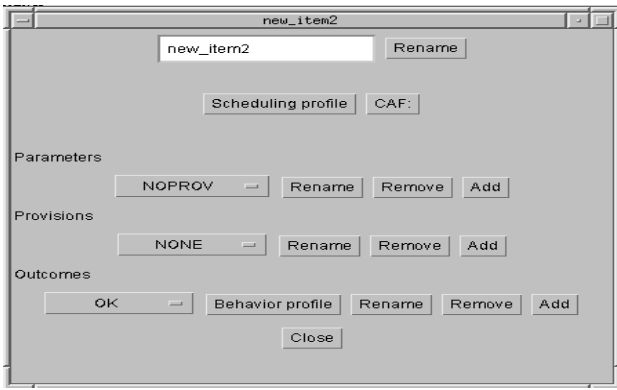
Typical GUI selection boxes guide the creation of new items, and the same constructs are used to edit existing items. The first thing users do is to create the individual items in the plan. Users are given a choice of four types to create: task, action, non-local task, or library. Library is not released yet. If a wrong selection is made, the item must be deleted and added anew. To create new elements of the plan, click on “edit” and “add item” from the PlanEditor.

### 2.2.2 Item “Decorations”

At any point, an item can be edited by clicking on “edit” and “edit item” to produce a screen like Figure 3. It also pops up automatically when an item is created. This is where parameters, provisions, and outcomes of the items are specified.

### 2.2.3 Characteristic Accumulation Function

DECAF allows user control of the Characteristic Accumulation Function (“CAF”) associated with a Task. Four choices are available:



**Figure 3: Edit or Create Item Attributes**

- **and** requires that all sub-tasks be completed before the task completes,
- **or** requires that *at least* one sub-task be completed before the task completes,
- **xor** requires that *at most* one sub-task be completed before the task completes, or
- **sum** chooses sub tasks so that the maximum value is attained.

Use of the CAF, scheduling profile, and behavior profile are beyond the scope of this paper. These are used in Design-to-Time and Design-to-Criteria scheduling [4].

#### 2.2.4 Lines – When and How to Draw Them

Lines are of two types in plans: those that represent messages, and those that represent task “ownership.” (The proper term is “reduction” because we reduce a plan to its steps.) Both types of line are placed into the plan by clicking on the source of the line with the **center** key of a 3-button mouse, then clicking again, with the same key, on the destination point of the line. Erase a line by repeating this process – click on the origin and then click on the destination. The origin for an ownership line is the Task.

Communication lines (blue lines on the screen) indicate that the output message from one item is to be the input item to another item. For example, in Figure 2 the output message associated with a return of “OK” from the routine “queryDB” will be sent to the agent indicated in that message (that is, into the cloud). The reply message from that agent (from the cloud) is the reply message that the Task `sdbwAsk_all` returns to its invoking task `Demo_ask_all`, and is thus the reply that this task will provide when it is invoked. Outputs can be sent to two different places concurrently by drawing lines from the output to two different destinations.

Ownership lines indicate the range of tasks and actions over which the Characteristic Accumulation Function of a Task will range. For example, in the demonstration SDBW plan, the “`sdbwAsk_all`” task is not complete until both of the tasks that it “owns” have been completed. This example is a bad one to explain the use of the characteristic function – the current example is too simple. Note that a task will not schedule actions that it does not own, and actions that have no task owning them cannot be sent messages.

### 3. MESSAGES IN DECAF

There are eight fields to a DECAF message:

- **performative** is the standard KQML or FIPA communicative action the receiving agent is to perform. If the action is not standard, use “achieve” and add a “:task xxx” specification into the content (see below) field of the message.
- **sender** indicates the name of the sending agent.
- **receiver** indicates the agent to receive the message.
- **reply-with** is a string that the sending agent directs the receiving agent to put in the “in-reply-to” field when it, the receiving agent, is sending a reply to the message currently being sent.
- **in-reply-to** is the above reply-with string from a prior message. (Null in initiating tasks.) It is appropriate to think of this and the above field as specifying and using a token which aids sending and receiving messages in a conversation. This token identifies which conversation is referenced, and the agent initiating an exchange of messages gets to select it (with “reply-with”).
- **language** is the language in which the message is expressed. This is “DECAF” or is domain dependent.
- **ontology** is the application-specific, user-determined ontology used by the receiving agent to identify that the sending agent understands the ontology used on the receiving end of the message.
- **content** is the application-specific information to be conveyed by the message. This is domain dependent, but often follows the <:keyword,value> conventions of language=“DECAF”, allowing standard DECAF tools to manipulate the contents of the content field.

Here is example code that sets these values:

```
public ProvisionCell queryDB(LinkedListQ Plist, Agent Local)
{
    KQMLmsg K = new KQMLmsg();
    K.addFieldValuePair("performative", "ask-all");
    K.addFieldValuePair("sender", Local.getName());
    K.addFieldValuePair("receiver", "agentName");
    K.addFieldValuePair("reply-with", "REPLY-TOKEN");
    K.addFieldValuePair("in-reply-to", ""); // not a reply
    K.addFieldValuePair("language", "DECAF");
    K.addFieldValuePair("ontology", "Demo");
    K.addFieldValuePair("content", ":keyword value");
    return new ProvisionCell(K.getKQMLString(), "OK");
}
```

Note that we use the Agent function “`getName()`” to insert the sending agent’s name into the message. The above sends an “ask-all” message to an agent that understands the “Demo” ontology. The sending agent’s invocation of DECAF will send the message to the agent designated by “agentName” and that receiving agent’s invocation of DECAF will start a task with the name “Demo\_ask\_all” when it receives the message. DECAF appends the performative to the ontology to determine the Task name that it will invoke. If the performative is “achieve” DECAF will append the value of “:task” in the content field to the ontology to identify the name of the class to be invoked. Note: to send a message to `_Shutdown`, the ontology field must be null. Also, note that DECAF demotes “\_” to “.” in names – while FIPA and KQML require “\_” in communicative actions, Java prohibits them, thus requiring this conversion.

## 4. JAVA CODE FOR DECAF AGENTS

Here we offer little advice – we do not know what users want the agents to do. Modified code from the SDBW agent (from the demonstration) serves as a partial example of the code needed. This code includes manipulations of the GUI within SDBW. Observe the following in the code below:

- In the window closing event, there is a reference to “local.send(msgString)” when the window is closed. This send function sends a message, but can accept no replies. Use this feature with extreme caution (contrary to the example).
- To understand the contents of the advertising messages, refer to [3].
- The return from “formatAdv” is “OK” (upper case) while one of the returns from “Response” is “ok” (lower case). DECAF is case sensitive.
- The “message” portion of the return from Response is “SOME STRING HERE”, which is meaningless (because the receiver needs to know only that the task was done). There must be something specified in this slot for DECAF to work. This is how to meet these seemingly contradictory conditions.

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import java.awt.event.*;

public class sdbwStartup
{static JTextArea ta;
  LinkedList P;
  Agent local;
public sdbwStartup()
{ }
public ProvisionCell formatAdv(LinkedList Plist,Agent Local)
{ String message = new String(Util.getValue(Plist, "MESSAGE"));
  KQMLmsg K = new KQMLmsg();
  Local.userHash.put("Gatekeeper", "Keymaster");
  try { UIManager.setLookAndFeel(
    UIManager.getCrossPlatformLookAndFeelClassName());
  } catch (Exception e) { }
  JFrame f = new JFrame(Local.getName());
  ta = new JTextArea("Starting Agent...",25, 5);
  ta.setEditable(false);
  JScrollPane jsp = new JScrollPane(ta,
    JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
    JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
  P = Plist;
  local = Local;
  f.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      //send a Shutdown message to the agent
      KQMLmsg Q = new KQMLmsg(Util.getValue(P, "MESSAGE"));
      Q.addFieldValuePair("sender",Q.getValue("receiver"));
      Q.addFieldValuePair("receiver",Q.getValue("receiver"));
      Q.addFieldValuePair("content", ":task Shutdown");
      local.send(Q.getKQMLString());
    }
  });
  f.getContentPane().add(jsp);
  f.setSize(400,300);
  f.setVisible(true);
  K.addFieldValuePair("performative", "advertise");
  [ ... See Section 6.2 for snipped code ... ]
  ":keywords Information Extraction");

  return new ProvisionCell(K.getKQMLString(),"OK");
}

public ProvisionCell Response(LinkedList Plist,Agent Local)
{ String message =
  new String(Util.getValue(Plist, "MESSAGE"));
  KQMLmsg K = new KQMLmsg(message);
  System.out.println(K.getKQMLString());
}
```

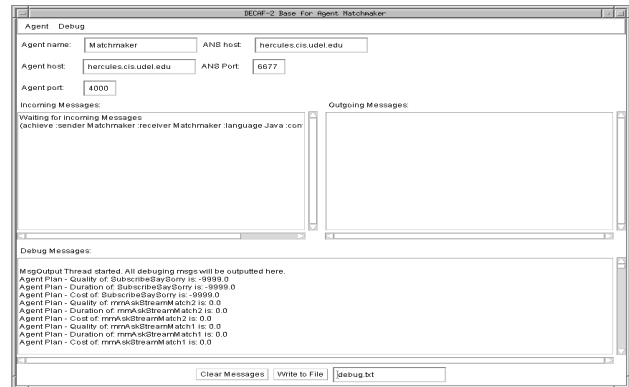


Figure 4: DECAF Base Agent Screen

```
if(K.getValue("performative").equals("error"))
{ try {
  Thread.sleep(10000);
} catch (Exception e) {}
K.addFieldValuePair("performative", "advertise");
[ ... See Section 6.2 for snipped code ... ]
"Information Extraction");
return new ProvisionCell(K.getKQMLString(),"fail");
}
else
{ return new ProvisionCell("SOME STRING HERE","ok");}
}
```

## 5. DECAF'S AGENT GUI

DECAF agents can be started using a programmer's GUI. This GUI is by-passed when parameters are provided on the command line. Normally, the GUI is only used when debugging because it provides invaluable information on the messages the agent has sent and received, and upon the general state of the agent. The command bar in Figure 4 shows choices for controlling the base agent screen. The debug button reveals a list of choices of what type of debug messages will appear in the bottom half of the base agent screen. Normally, all are turned on, so all of the pieces of DECAF will write messages to the screen. If this is overwhelming for a particular task (or too time consuming, as this is actually a very slow thing to ask DECAF to do), the less currently interesting parts of DECAF can be muted by clicking on them in the Debug list. Often, for beginners, only the Scheduler needs to show its debug messages.

The Scheduler reveals in the debug messages tasks and actions that have been scheduled for receiving execution time. If, over the course of executing an agent it slows down, it may be that it inadvertently created tasks that never got completed – these then take up space and need to be checked repeatedly to see if they are enabled to be run.

An agent with “Size of Tasks: 0” at the bottom of the screen is simply waiting for a message. If that is not the intention, then some activating message has not been sent to the agent. Interpretation of the contents of the messages is very domain dependent, precluding additional useful discussion here.

The first item on the command bar is “Agent” and has four choices on it. Choosing “Exit” shuts down the agent – not just the GUI, but it **shuts down the whole agent**.

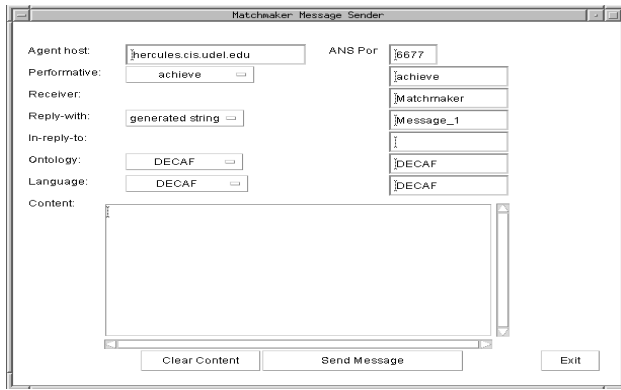


Figure 5: Agent Messaging Screen

Using this option when done with the agent will, in general, shut down the agent normally. If, however, there are jammed threads in the agent, this exit button may not shut the agent down properly. Then the agent must be killed with Cntl-C or other system command. If the agent is killed (i.e., the Exit does not work), use ANSQuery to remove the name of the agent from ANS.

## 5.1 DECAF KQML Message Sender

To send messages with the GUI, choose “Agent” and then “KQML Message Sender” to get the message sending screen Figure 5. This screen lets users send messages “manually” to agents simply by filling in the needed parts and pressing the “Send Message” button.

If a user sends the message correctly, it appears in the “Outgoing messages” portion (right hand side) of the base agent screen of the sending agent. If the message is received by an agent, it appears in the “Incoming messages” portion (left hand side) of the base agent screen of the receiving agent. If an agent sends a message to itself, the message will appear on both screens. Note that Figure 4 contains an incoming message – this is the message that DECAF sends to the agent to Startup.

Sending messages must be done with care. The following fields *must* be filled:

- **Performative**, which may be the default of “achieve”.
- **Receiver**, so DECAF knows where to send it.
- **Ontology**, so the receiver can compose task names.
- **Content** with information for the receiving agent (and which cannot be empty).
- **Language** is DECAF or Java in most situations.

Most of the time, it is best to let the agents send the messages themselves. The message sender is most helpful to isolate aberrant behavior in an agent, especially when the receiving agent also has a GUI. Messages sent in such a situation, coupled with print statements in the receiving agent, can reveal the most interesting behavior on the part of agents. Another use for the message sender is to debug stand-alone agents, or to see the response of a particular agent to a particular message.

## 6. SOME USEFUL DECAF METHODS

The code examples shown above reference several methods within the DECAF structure. Here we discuss useful DECAF methods in a little more detail. Our discussion covers the Agent methods available through the Agent object passed to methods invoked by DECAF, and a selected few that are available through the Util and KQMLmsg objects (the messages DECAF uses). The methods are listed under the Java code that defines them.

### 6.1 Agent Methods

Consider this code snippet from the code examples above:

```
public ProvisionCell queryDB(LinkedListQ Plist, Agent Local)
{ String MyNameIs = Local.getName();
  . . .
}
```

It resembles every method that DECAF invokes because it shows the arguments passed to the method: a LinkedListQ and an Agent structure. Messages are in the LinkedListQ and agent information in the Agent structure. This section deals with methods in the Agent structure. The next section covers KQMLmsg structure methods.

#### 6.1.1 String getName()

The Agent method “getName()” simply returns the String that is the name of the agent. For example, the string returned for the Matchmaker will always be “Matchmaker”. This is useful for multi-agent systems where there are several invocations of the same agent code, and each agent needs to know the name under which it was invoked.

#### 6.1.2 void send(KQMLmsg k)

Normally this method is *not* used, as the “cloud” construct is the preferred procedure and makes messages visible in the agent plans. This method sends a well-formed message. Note that there is no way to handle responses to such methods in the method that sends them, and also note that these messages are outside the messages diagrammed on the DECAF plan files. Suggested for use only by advanced DECAF agent programmers.

#### 6.1.3 void DebugAgent(String Message)

Puts the indicated Message onto the output screen of the base agent GUI screen. Useful when debugging agents with particularly intricate interactions and structures. It may be more appropriate to consider simplifying the structure and interactions of the agent, as well.

## 6.2 KQMLmsg Methods

This code snippet is enhanced from the demonstration:

```
public ProvisionCell queryDB(LinkedListQ Plist, Agent Local)
{KQMLmsg Received=new KQMLmsg(Util.getValue(Plist,"MESSAGE"));
String OriginalSender = Received.getValue("sender");
String OriginalReplyWith= Received.getValue("reply-with");
String categ = Received.getContentFieldValue("category");
if(categ.equals("")) categ = new String("CategoryMissing");
. . .
KQMLmsg K = new KQMLmsg();
K.addFieldValuePair("performative", "advertise");
K.addFieldValuePair("sender", Local.getName());
K.addFieldValuePair("receiver", "Matchmaker");
K.addFieldValuePair("reply-with", "nothing");
K.addFieldValuePair("language", "DECAF");
K.addFieldValuePair("ontology", "Matchmaker");
K.addFieldValuePair("content",
":performative ask-all " +
```

```

        ":ontology Demo " +
        ":language DECAF " +
        ":taskName ask_all " +
        ":parameters parka " +
        ":keywords Information Extraction");
    return new
    ProvisionCell(K.getKQMLString()+Util.addTimeout(123),"OK");
}

```

with the functions of the methods described below. The messages are accessed via a utility function `Util.getValue` that is applied to the linked list queue associated with the task. The MESSAGE designation always works, and is useful for retrieving messages sent to Tasks that have not had parameters nor provisions specified. For the SDBW example shown in Figure 2, the term “parka” is used, and the message is also accessible as “parka”.

### 6.2.1 String `getValue(String field)`

This method returns the string that follows the designated field in the message. If the designated field is “sender”, then the string returned is the string that follows “.sender” and precedes the next “.” in the message stream. The message stream itself is a string, with the colons used both to signal the end of a string and to indicate keywords. Do not include a colon in the name sent to the function (i.e., “sender” and not “.sender”), and only request one of the eight fields that constitute a KQML message.

### 6.2.2 String `getContentFieldValue (String field)`

This method returns the string that follows the designated field in the the content field of the message. If the designated field is “category”, then the string returned is the string that follows “.category” and precedes the next “.” in the message stream. The contents of the content field is a string, with the colons used to indicated keywords. As with message fields, do not include a colon in the name sent to the function, and only request one of the fields that you have required to be included. If you ask for a field that is not required (by virtue of being a parameter or provision), you should test for a null value and replace it with something more useful, as is done in the above example for “category”.

### 6.2.3 `addFieldValuePair(String field,String value)`

This is used to set any of the eight fields of a KQML message. Note that the content field must be set in one such command, with the value either built in the call (as above in the code example) or with a string that has been separately constructed. A common error in building the string is to omit spaces between arguments, resulting in their concatenation by Java. The above example places the spaces at the end of each section of the content field that it builds. A second common error is to believe that this command concatenates content information when the field name is “content”. This `addFieldValuePair` replaces the value that is in the message with the stated value, in a manner that some would say would require the name of the function to be `setFieldValuePair`. Its name is `add`. Its function is to `set`.

Note also that DECAF does not ensure that only one of the eight approved fields is entered. If other values are used, the insertion will occur, with unspecified results. The most likely problem will be that information meant for the content field has been inadvertently placed as a ninth (or subsequent) entry in the KQML message, making it unavailable

to receiving agents that (properly) look in the content field for such information.

### 6.2.4 String `getKQMLString ()`

This converts a `KQMLmsg` into a string for use by DECAF in sending outcomes as messages. The reason for this is because of the need, within methods, to work with portions of the KQML messages. That need dictates that messages be taken apart upon receipt, manipulated by the agent methods, then reassembled before sending the messages along. This method (`getKQMLString`) does the reassembly into a string. It is usually employed as one of the two following examples. The first will wait forever to receive a response, the second will wait 321 seconds. The `addTimeout` method is discussed below, with other utility functions. `return new ProvisionCell(K.getKQMLString(),"OK");`  
`return new ProvisionCell(K.getKQMLString() + Util.addTimeout(321),"OK");`

## 6.3 Utility Methods

Some useful methods are located in the aptly-named “Util” class. These include the method that sets the amount of time to wait for a reply to a message and the method that retrieves individual messages, parameters, and values from the `LinkedListQ` structure.

### 6.3.1 String `getValue(LinkedListQ List,String Name)`

Returns as a string the contents of the `LinkedListQ` that has the indicated name. This feature is in general used to extract the source message for a task. The code snippet discussed under `KQMLmsg Methods` contains an example of its use to retrieve the MESSAGE that triggered the Task.

### 6.3.2 String `addTimeout(int time)`

This method is used to append a time limit to a message. If the agent to which the message has been sent does not reply within the stated number of seconds, DECAF will generate and deliver an error message to the sending agent (i.e., the one that has added the timeout) on behalf of the intended receiving agent. A timeout has been artificially inserted into the code shown above to demonstrate the use of the feature. The message to the Matchmaker will timeout after 123 seconds. If the user does not care whether a message gets where it's supposed to go, a timeout of zero is used, and there is no waiting for a response.

## 6.4 The Key to Programming DECAF Agents

The Agent has attached to it a Java hashtable for the user to save persistent information. The sample code contains a one line example of its use. This Hashtable can be used to store all the objects that you want the agent to have access to. It is defined in the Agent class with the statement:  
`public static Hashtable userHash=new Hashtable();`

## 7. BREAKING THE DEMONSTRATION

The demonstration code can be run in five different windows to demonstrate the actions of each agent. This shows what actions each agent performs, and also mimics what would happen if the five agents were all on different machines (which DECAF supports). Also, it allows the code to be “broken” while it previously appeared to be correct. Breaking these agents is done by starting the agents in the wrong order. It is a **bad thing** that these agents can be

made to fail by starting them in the “wrong” order. Agents should not have to depend on the order in which they are started in order to work successfully. Agents should be robust to conditions over which they have no control. If the needed other agents are not available, the needing agent should temporize, improvise, or do without. Agents must be robust to function properly.

## 7.1 Start Matchmaker After SIA

The Java error message that appears means that SIA failed while parse the response it got. Unfortunately, SIA is not parsing a successful response from the Matchmaker – SIA has received an error message from DECAF saying that the intended receiver of the message (the Matchmaker) does not exist. SIA does not have a way of dealing with that condition, so SIA has become totally confused about where it is and what it is doing. SIA is “jammed,” and this jam is unrecoverable by the user. Note four things about this jammed state:

- This is only one thread that is jammed. If the agent had other threads going, they would continue to operate, and, if they relied on this thread, their behavior would become “unexplainable.” The odd behavior is the agent designer’s fault.
- With many threads in the same window (unavoidable), the above error message might not be visible on the output window, having been scrolled out of sight by subsequent messages from other threads.
- Java error messages contains no information about the agent that originated it. Operating several agents in one window (avoidable) can make debugging difficult if the error message is the only clue you have that something has gone wrong. Suggestion – write locational information to the screen frequently in the early stages of agent development.
- The agent can be terminated with a Cntl-C command. If an agent is terminated this way, the terminated agent does not *unregister* with the ANS. (This may be a Java issue.) You should then use ANSQuery to perform the unregistering. Failure to unregister the agent prevents it from registering with that name again, unless the ANS has also been restarted.

## 7.2 Start SDBW after Matchmaker and SIA

This is the most interesting of the “errors” within the demonstration. All of the agents will work as planned, except there can be no successful queries to SDBW from SIA. The following is what has gone wrong: SIA queried Matchmaker before SDBW advertised, and since SIA does not query the Matchmaker a second time, SDBW’s advertisement comes too late to be received by SIA. Attempts to query databases that exist fail simply because there is no existing databases that are known to SIA. The interesting part is that you can see the SDBW agent, start other agents that can send and receive messages involving SDBW, and so on, yet the SIA fails and DECAF looks broken. SIA should have used “subscribe”.

By far, this is the most popular way to break DECAF agents – interfere with the sequencing and timing of messages. The solution is to design agents that can’t have their message sequences or timings interfered with – truly the sport of royalty.

## 7.3 Concurrency

DECAF, based on Java threads, creates opportunities for the benefits and problems of concurrency. Tasks, Actions, and Clouds can all have multiple instantiations, and user Java code must handle concurrency in Tasks and Actions. Multiple Tasks come in to existence because DECAF starts a new Task (a new instance of the appropriate Task class) each time the provisions and parameters for a Task are satisfied, and the Task is enabled. Multiple Actions come in to existence because DECAF starts a new Action for each enabled action. The ability to easily begin such multiple threads, without the need to explicitly construct them within agents, is a DECAF strength – programmers can then concentrate on what the agent does rather than on how to accomplish the required threading and communication. This, in turn, allows agents to do useful things with less programming. DECAF’s Semaphore class (not mentioned above) and the Hashtable available to the user are available to resolve issues of deadlock prevention and mutual exclusion. And the DECAF GUI is available to assist in the debugging.

## 8. CONCLUSION

DECAF provides high-level language support and pre-built middle-agents for building multi-agent systems across networks. By taking an “agent operating system” approach rather than the “API” (Application Programming Interface) approach to creating agents, DECAF has been shown to allow students without high levels of expertise to get to the “interesting” parts of a multi-agent system more quickly. The material presented in this paper, together with the demonstration code and other DECAF reference materials, provides the interested researcher with sufficient material to implement DECAF agents that provide useful services.

Work continues on expanding capabilities within DECAF. Re-usable libraries of tasks will allow the employment of developed task capabilities across several agents, multiplying the benefit of building robust agents with generic capabilities (such as advertisers, local information caches, and information gathering agents). Full centralization of agent reporting and management has applications in a number of domains, and is under development. Additional planning and scheduling capabilities within the structure are also underway.

## 9. REFERENCES

- [1] J. Graham and K.S. Decker. Towards a distributed, environment-centered agent framework. In N.R. Jennings and Y. Lesperance, editors, *Intelligent Agents VI*, LNAI-1757, pages 290–304. Springer Verlag, 2000.
- [2] John R. Graham. *Real-Time Scheduling in Distributed Multi-Agent Systems*. PhD thesis, University of Delaware, 2001. See <http://www.cis.udel.edu/~decaf/main.pdf>.
- [3] Mikko Laukkanen and Jukka Eskelinen. Requirement Specification for the DECAF-Matchmaker. Technical report, University of Delaware, May 1999. Updated June 2000 by Foster McGearry.
- [4] T. Wagner, A. Garvey, and V. Lesser. Complex goal criteria and its application in design-to-criteria scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Providence, July 1997.