DECAF Programming: An Introduction

Foster McGeary

Computer and Information Sciences
University of Delaware

{mcgeary@cis.udel.edu}

April 9, 2001

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.

John R. Graham, Ph.D., completed DECAF before departing to teach at Coastal Carolina University. Dr. Graham's new website is http://www.coastal.edu/~graham/. DECAF continues as a part of Dr. Decker's research program.

This paper is a brief introduction to using DECAF for simple agent systems. It borrows liberally from the documentation prepared by Drs. Graham and Decker or under their direction. That documentation is available directly at http://www.eecis.udel.edu/~decaf/, the DECAF website at the University of Delaware. Interested persons are encouraged to visit that website and to use (and download if helpful) the material available there.

This material is based upon work supported by the National Science Foundation under Grant Nos. IIS-9733004 and IIS-9812764.

Contents

1 DECAF - Distributed Environment Centered Agent Framework

DECAF (Distributed Environment Centered Agent Framework) 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, execution monitoring, coordination, and eventually learning and self-diagnosis. 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 a graphical user interface called the Plan-Editor. In the Plan-Editor, executable actions are basic building blocks that can be chained together to achieve large, complex goals in the 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. We do not address these advanced uses in this paper.

1.2 DECAF Goals

The goals of the architecture are to provide a modular platform suitable for research activities, allow for rapid development of third-party domain agents, and provide a means to quickly develop complete multi-agent solutions using combinations of domain-specific agents and standard middle-agents and to take advantage of the object oriented-features of the Java programming language. DECAF distinguishes itself from many other agent toolkits by shifting the focus away from the underlying components of agent building such as socket creation, message formatting, and the details of agent communication. In this sense DECAF provides a new programming paradigm: Instead of writing lines of code that include system calls to a native operating system (such as read() or socket()) DECAF provides an environment that allows the basic building block of agent programming to be an agent action. DECAF is an agent operating system. Code within agents can make calls to DECAF to send messages, search for other agents, or implement a formally specified coordination protocol. The interface to the framework is a limited set of utilities that remove, as much as possible, the need to understand the underlying structures. Thus, the programmer does not need to understand Java network programming to send a message, or learn Java database functions to attach to the internal knowledge base of the framework.

1.3 DECAF Architecture

Figure 1 shows the structure of DECAF.

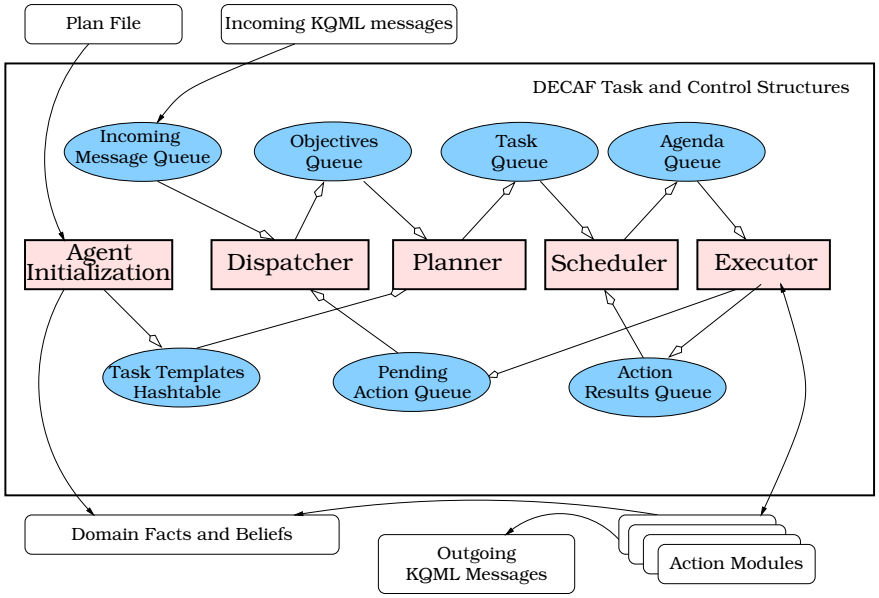

Figure 1: DECAF Architecture Overview

Concern in this document 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 you will build have about the domain in which you are solving problems with DECAF. Please consult the references at the end of this paper for information about the insides of the box. The next section offers some insights into the box in the center, but may be skipped on an initial reading.

1.4 Broad Outline of the DECAF Architecture

As shown in Figure 1, there are five internal execution modules (square boxes) in the current DECAF implementation, and seven associated data structure queues (rounded boxes). DECAF is multi-threaded, and thus *all modules execute concurrently, and continuously* (except for agent initialization).

Agent Initialization.

When an agent is started, the *Agent Initialization* module will read a plan file containing basic action definitions and pre-defined task network reductions. Each task reduction specified in the plan file will be added to the *Task Templates Hashtable* (plan library), indexed by the particular goals that the reduction achieves.

The agent initialization process also attempts to achieve a *Startup* goal. The Startup tasks of a particular agent might, for example, build any domain data/knowledge bases needed. Startup tasks may assert certain continuous maintenance goals or other initial achievement goals for the agent. The last thing the Agent Initialization Module does is register with an Agent Name Server and set up all socket and network communication.

Dispatcher.

The Dispatcher waits for incoming KQML messages which will be placed on the *Incoming Message Queue*. An incoming message contains a KQML (or FIPA) *performative* and its associated information. An incoming

message can result in one of three actions by the dispatcher. First, the message may be a part of an ongoing conversation. In this case the dispatcher will find the corresponding action in the *Pending Action Queue* and provide the incoming message as an enabling provision (see "Parameters and Provisions", below). Second, a message may indicate that it is part of a new conversation. If so, a new *objective* is created (equivalent to the BDI "desires" concept[9]) and placed on the *Objectives Queue* for the Planner. An agent typically has many active objectives, not all of which may be achievable with an agent's limited time and resources. The last thing the Dispatcher is responsible for is the handling of error message replies to malformed messages.

Planner.

The Planner monitors the Objectives Queue and plans for new goals, based on the action and task network specifications stored in the Plan Library. A copy of the instantiated plan, in the form of an HTN corresponding to that goal is placed in the *Task Queue* area, along with a unique identifier and any provisions that were passed to the agent via the incoming message. The Task Queue at any given moment will contain the instantiated plans/task structures (including all actions and subgoals) that should be completed in response to all incoming requests and any local maintenance or achievement goals. A graphical plan-editor GUI allows the construction of static HTN planning task structures.

Scheduler.

The *Scheduler* waits until the Task Queue is non-empty. The purpose of the Scheduler is to determine which actions *can* be executed now, which *should* be executed now, and in what order.

For DECAF, the traditional notion of BDI "intentions" as a representation of a currently chosen course of action is partitioned into three deliberative reasoning levels: planning, scheduling, and execution monitoring. This is done for the same reasons given by Rao [9]—that of balancing reconsideration of activities in a dynamic, real-time environment with taking action [10]. Rather than taking the formal BDI model literally, we develop the deliberative components based on the practical work on robotics models [11]. Each level has a much tighter focus, and can react more quickly to external environment dynamics than the level above it. Most authors make practical arguments for this architectural construction, as opposed to the philosophical underpinnings of BDI, although roboticists often point out the multiple feedback mechanisms in natural nervous systems.

Once an action from the Task Queue has been selected and scheduled for execution, it is placed on the *Agenda Queue*. In a very simplistic Scheduler, the order might be first-come-first-served (FCFS). Recent work has resulted in the development of a sophisticated *design-to-criteria* action scheduler [12] that efficiently reasons about action duration, cost, result quality, and other utility function characteristic trade-offs. This scheduler is also available for use in DECAF agents.

Executor.

The *Executor* is set into operation when the Agenda Queue is non-empty. Once an action is placed on the queue the Executor immediately places the task into execution. When the action completes it signals a specific action outcome, and the result is placed on the *Action Result Queue*. The framework waits for results and then distributes the result to downstream actions that may be waiting in the Task Queue. Once this is accomplished the Executor examines the Agenda queue to see if there is further work to be done.

1.5 Goals of this Documentation

This documentation is intended to assist the user unexperienced with DECAF to "get up and running" with simple agents. Eventually, DECAF usage requires familiarity with Java. No Java instruction is provided here. This documentation is intended for someone who is either familiar with Java, or who knows enough C++ to fake it convincingly well. Users are presumed to be self-teachers who are interested in a few helpful hints.

With that in mind, we proceed.

2 Step One - Experience the Demonstration

DECAF has a demonstration accessible through the DECAF Homepage (http://www.eecis.udel.edu/~decaf) by clicking on the "Downloads" button on the upper left hand side of the page. After registering as a DECAF user, click on the highlighted "download page" to see the then-current documentation for the demonstration. Do whatever portion of the instructions you are comfortable with. As of this writing, the download will take about three megabytes of space for the demonstration software. We ask that you access and download the demonstration – it was prepared for teaching purposes.

2.1 Notes on the Demonstration

Operational problems with the demonstration can be created by not following the directions correctly. Following the demonstration properly produces a five-agent world:

1. **ANS** is the Agent Name Service, which is *essential* to have running for any DECAF agent application to find the addresses of other agents.

2. **ANSQuery** is an optional, useful but cycle-consuming, application that lists the names that ANS has assigned to agents that requested them.

3. **Matchmaker** is optional, but the demonstration application uses it. Matchmaker accepts "advertisements" from agents and provides the names of advertising agents to any agents that request such names.

4. **SDBW** is a Simple Data Base Wrapper that advertises with Matchmaker. It advertises itself as (See [7] for a discussion of the Matchmaker.) a provider of services, and Matchmaker stores the keywords by which it would like to be found.

5. **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.

2.2 Demonstration

The remainder of this paper assumes you have downloaded and worked with the demonstration. It would now be helpful to obtain the demonstration and exercise it. After doing so, you can then profitably continue with this paper.

2.3 Breaking the Demonstration

Okay, now let's break the demonstration. First, though, you must close down the demonstration you have and ensure that all the agents are removed. It would be a good idea to terminate the ANSQuery agent last, as it will let you know if you have terminated the other agents correctly.

Then open five windows on your machine, with the intention to run each of the agents in a different window. This will allow you to see 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 us to "break" code that has appeared to be correct.

We now review the three ways to "break" the demonstration code.

2.3.1 Forget to Set the CLASSPATH Variable Properly Set

This is sort of an easy, practice, error condition to create. It prevent agents from accessing the code they need to run. It's a simple error, and easy to perform.

2.3.2 Start the Agents in the Wrong Order

There are several wrong orders to start these demonstration agents, but they are not unlimited, and each has a key to making the demonstration fail. Let's identify them after making the following very important point . . .

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. If you've had courses in finite state automata, now is the time to apply that knowledge. Agents must be robust to function properly.

Don't Start ANS First

This breaks the demonstration because other agents can't even get started – they absolutely require the ANS to run. Notice the lack of error messages when you try to start, for example, the Matchmaker without previously having started the ANS. The ANS is so fundamental to DECAF that DECAF becomes totally befuddled without it, and doesn't know what to say or do. So it says nothing and does nothing.

So, if you try to start agents and they won't start, check to see if ANS is running. The easiest way to do this is to start ANSQuery. Since there is little that the user can do to confuse ANSQuery, this is a good test. If ANSQuery fails to find an ANS, it will repeat the message

```
ANS connection failed.
ANS connection failed.
ANS connection failed.
ANS connection failed.
ANS connection failed.
ANS connection failed.
ANS connection failed.
ANS connection failed.
ANS connection failed.
ANS connection failed.
ANS connection failed.
```

forever.

As soon as ANS starts, ANSQuery stops this incessant whining and functions normally.

Start Matchmaker After SDBW

If you do this, then SDBW cannot place its advertisement. Without the advertisement, SIA cannot find SDBW. However, someone has recently modified the SDBW to prevent this little game. SDBW now has a loop in it so that it, like ANSQuery, will continually try to advertise with the Matchmaker until its advertisement is accepted. (We'll show you how to do the looping later in this paper. Use of planeditor to view SDBW.lsp will show the plan file that achieves the looping.)

Start Matchmaker After SIA

This works – it breaks the demonstration. SIA produces the following code:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Startup Zero constructor % <==============
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% findRMDBs Zero constructor % <==============
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
java.lang.NumberFormatException:
        at java.lang.Integer.parseInt(Integer.java, Compiled Code)
        at java.lang.Integer.<init>(Integer.java, Compiled Code)
        at Demo_findRMDBs.UpdateRMDBs(Demo_findRMDBs.java, Compiled Code)
        at java.lang.reflect.Method.invoke(Native Method)
        at ExecutorThread.run(ExecutorThread.java, Compiled Code)
```

The above Java error message means that SIA failed while executing the java language routine parseInt on and Integer. What is happening is that SIA is trying to 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:

1. 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" due to "an error in DECAF." No, the odd behavior is the agent designer's fault.

2. 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.

3. The above Java error message contains no information about which agent originated it. Operating several agents in one window (avoidable) can make debugging difficult if the above 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.

4. 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. You should then use ANSQuery to perform the unregistering for you. Failure to unregister the agent prevents it from registering with that name again, unless the ANS has also been restarted.

Restarting a Failed Agent

If you restart a failed agent that has not *unregistered* with the ANS, it will fail in the same manner as if there were no ANS. That is, you get no error message, and nothing appears to happen. Use ANSQuery to "highlight" and then "unregister" the name that the agent would like to use. DECAF takes care of the unregistering for agents that shut down normally (i.e., those that shut down by sending a "Shutdown" message to the agent).

Start SDBW after Matchmaker and SIA

This is the most interesting of the "errors" you can perform with the demonstration. All of the agents will work as planned, except you can make no successful queries with the 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.

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.

2.3.3 Change the Demonstration Code

The final way to break the demonstration is to change the DECAF plans or Java code that the agents use. Once in this realm, of course, its more fun to deal with your own agents. So, we now give up *entirely* on trying to break the demonstration, and move on to more constructive work. (Remember the techniques discussed here, though, in case you want to **break your own agent code.**)

3 Specifying 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 Plan Editor and Java. We will discuss the DECAF Plan Editor in the next subsection and then we will show some agent Java code. After that, you are on your own.

3.1 The Plan Editor

The DECAF software used is the Plan Editor. The Plan Editor is a graphical interface used to create the programming for your Agent. DECAF is of two minds when it comes to what to call the Plan Editor. When you execute it, you issue the command

```
java planeditor
```

and so sometimes you will see references to planeditor and sometimes to Plan Editor. They are everywhere the same piece of software.

Planeditor is the software that draws lines and boxes and clouds that specify to DECAF what your 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 Plan Editor as shown above while you are in the SDBW directory of the demo. Clicking on the standard "file" then "open" menu items will show you 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 causes Figure 2 to appear. This is the whole "plan file" for SDBW:

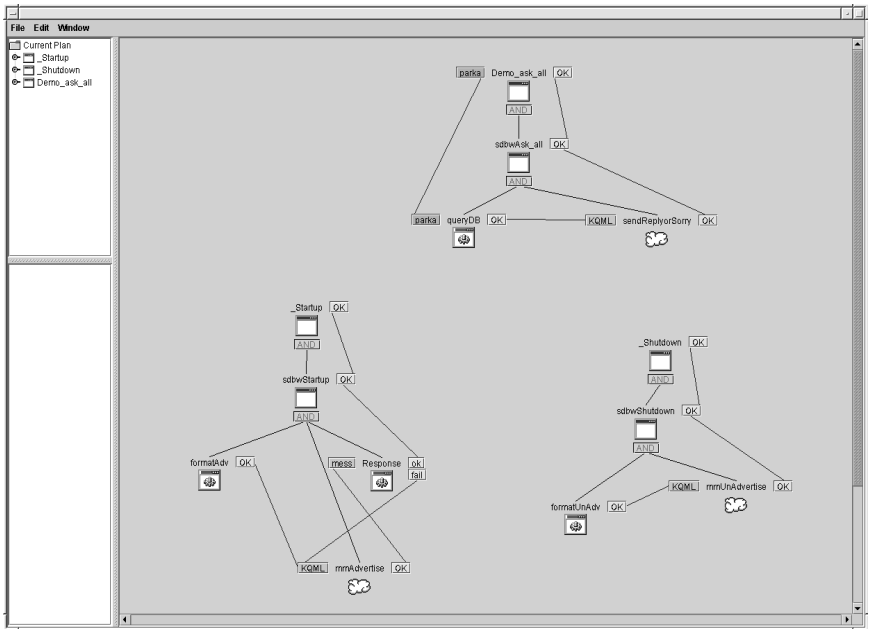

Figure 2: The Big Picture

3.1.1 Plans Look Like Forests of Trees

The Program is a tree-like structure (Hierarchical Task Network, or "HTN" for short). Figure 2 contains three such tree-like structures:

1. **Demo_ask_all** which is the Task for the SDBW to answer queries.
2. **_Startup** which initiates the SDBW agent, including registering with the Matchmaker.
3. **_Shutdown** which closes down the SDBW agent, including unregistering with the Matchmaker.

Note that there are no lines connecting the three structures. There is no need to connect them because they are separate entities 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.

Let's look at one Task Structure, the one for "Demo_ask_all" as shown in Figure 3.

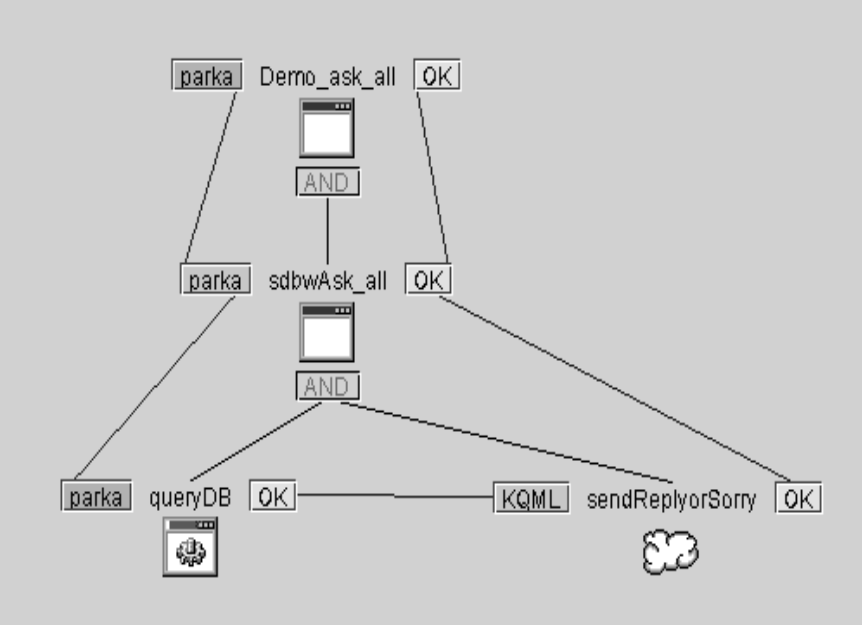

Figure 3: The Demo_ask_all Task Structure

3.1.2 Components of a Task Structure

Each tree represents one task structure and also usually represent one Java code file, implying that there is close to a one-to-one correspondence of DECAF Tasks and Java code files. (This is essentially true, with, of course, exceptions for when it is not true. We'll get to those minor exceptions.) The leaf nodes are actions that must have associated code and class files. Each node (leaf and non-leaf) has 0 or more inputs (provisions or parameters) and 1 or more outcomes. Figure 3 shows all the major features of a DECAF agent Task. Let's identify them:

1. **Top-level Tasks** are the highest level task abstracts, and may or not correspond to java classes (one of the exceptions we mentioned above). 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. Section 4 has additional insight on how this part of the naming works.

2. **Intermediate Tasks** are java classes 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 3, 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 3.2.3).

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. (The bar at the top is also a different color, but that does not show up in non-color presentations.) 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. An example follows.

4. **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 Task, Actions, and Clouds (non-local Tasks):

- Inputs to an icon (called "provisions" or "parameters") are the items on the left of the icon.
- Outputs from an icon are the items on the right of the icon.
- 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, you have to go back to these plans either after many months or, more importantly, when you are under duress because the plan is not working as intended (the silly thing is working as specified!). In this latter case, 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 your "gopher" class of agent will select a seat for you. As always, naming is hard to do well.

Names for provisions are fairly easy – your application will suggest them. Names for outcomes will need to match the corresponding field in your 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. Let me say that again, because it is *so* important – Names for outcomes will need to match the corresponding field in your 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.

3.1.3 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 (Section 5). The actual source code for sdbwAsk_all.java comes with the demonstration.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class sdbwAsk_all
{
   public sdbwAsk_all()
   {
      System.out.println("sdbwAsk_all Zero constructor");
   }

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

Note the following in and about the above:

- The class name is the same as the lowest level Task name in the plan.
- There is, and must always be, a 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 a two strings, in this case a KQMLmsg converted to a string and the string "OK".
- The contents of the string (the second element) in the ProvisionCell returned from the queryDB method match exactly the outcome box in the plan for the queryDB action (see Figure 3).

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 has if it had left the action through one outcome when you had intended another(thus making the agent "act funny").

3.2 Particular Details of Plan Creation

The planeditor is a straightforward GUI, and you can always experiment with it if you have questions. 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.

3.2.1 Creating a New Item

Figure 4 shows the first work screen you will encounter when you begin to design your own agents. The first thing you must do is to create the individual "item"s in the plan. You will be given a choice of four types to create: task, action, non-local task, or library. Library doesn't work yet. If you pick wrong, you will have to delete the item and add it anew. (See Section 3.1.2 for a discussion of these components.) To get to Figure 4, click on "edit" and "add item" from the planeditor.

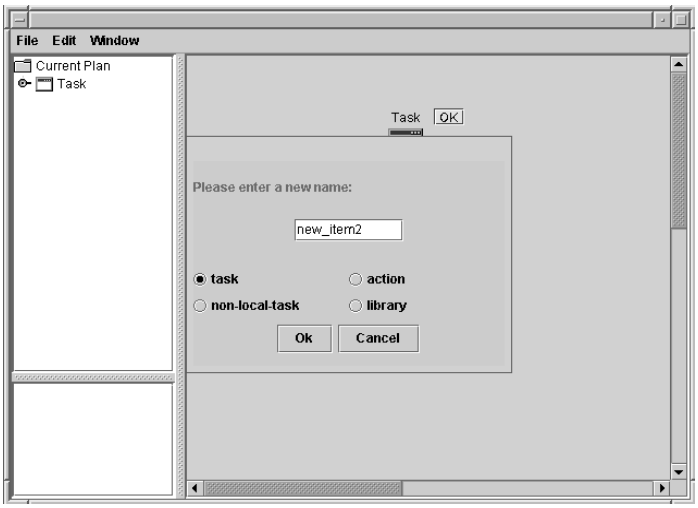

Figure 4: Initial Item Addition Screen

3.2.2 Item "Decorations"

At any point, you can edit an item by clicking on "edit" and "edit item" to produce a screen like Figure 5. It also pops up automatically when an item is created. This is where you specify the parameters, provisions, and outcomes of the items.

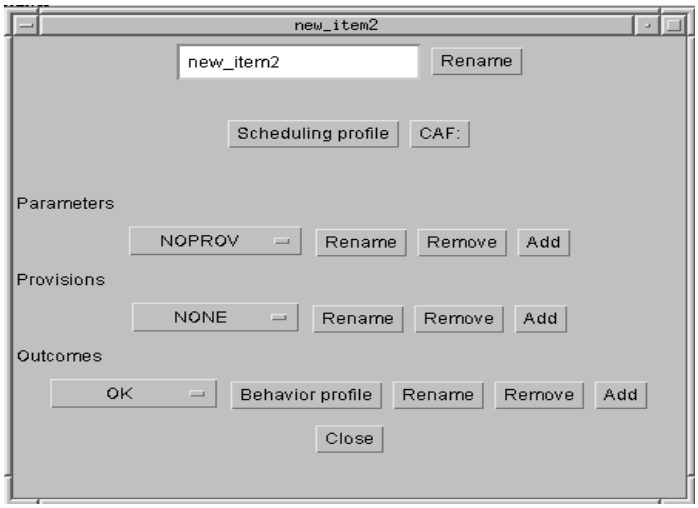

Figure 5: Edit or Create Item Attributes

3.2.3 Characteristic Accumulation Function

Figure 6 shows the screen that lets you control the Characteristic Accumulation Function ("CAF", but no relation to the CAF in DECAF) that is associated with a Task. The four choices let you choose whether DECAF will

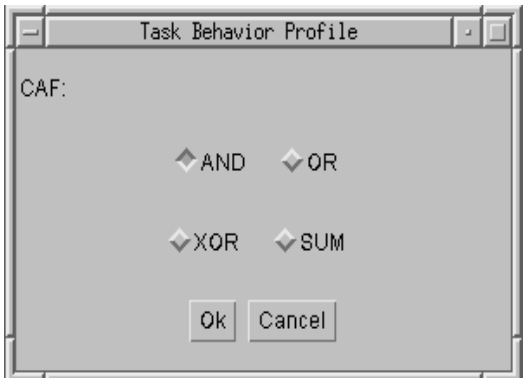

Figure 6: Characteristic Accumulation Function Choice

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

Use of the CAF is beyond the initial intention of the scope of this documentation. There is a little discussion of its use below, but beyond that you are urged to contact Dr. Decker directly.

3.2.4 Lines – When and How to Draw Them

Lines are of two types in plans: those that represent messages, and those that represent task (for lack of a better term) "ownership." (Okay, there is a proper term, and it 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. Interestingly, you can 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 (they will appear as blue lines on the screen) indicate that the output message from one item is to be the input item to another task. 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 Figure 2, 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. We will address it a little bit more below. 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.

3.3 A Simple Task Analyzed

Figure 7 shows a simple plan. This plan probably will not work, and we can profitably learn from it.

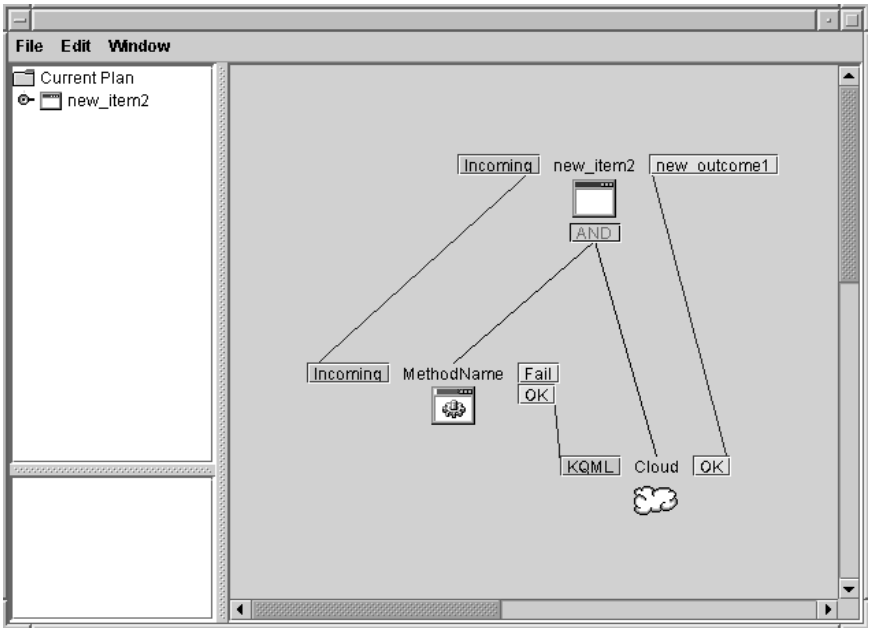

Figure 7: A Less Simple Task

The first, and easy, reason the plan might not work is that it is mis-named, or, at best, oddly named. DECAF tasks are reachable when the name they use is the concatenation of an ontology with a task name. The name used for the task shown in Figure 7 contains an underscore, as it would if there were a "new" ontology. In other words, this task is reachable only if the agent within whose plan it resides receives a message with the ontology "new" and the task set to "item2". Note that if the Task name had *no* underscore, it would be totally unreachable. The "null" ontology is used for Shutdown messages, which is why the top-level tasks for Startup and Shutdown begin with underscores.

Now for a slightly subtle reason for the plan to fail. Note the two possible outcomes for "MethodName" are a "Fail" outcome and the ever-popular "OK" outcome. If this agent exits MethodName via a return statement that tags the outcome as fail, the task will not terminate. There is no way to reach the "new_outcome1" outcome of the top-level task. Since this top-level task is not reached by a supporting outcome, DECAF will simply continue to wait for it to be satisfied, even if DECAF has to wait forever. This waiting for tasks that can never end tends to "clog" DECAF agents – tasks do not complete because the conditions for their completion are not met, generally because the task structure is not guaranteed to always produce an outcome. Section 6.1 briefly addresses how to use the DECAF GUI to assist in identifying this situation, should the agents you deal with create it.

4 Messages in DECAF

There are eight fields to a DECAF message:

1. **performative** tells the receiving what *standard, KQML/FIPA* action it is to perform. If the action is not standard, the value is set to "achieve" and a ":task xxx" specification is inserted into the content (see below) field of the message.

2. **sender** indicates to the receiving agent the name of the sender.

3. **receiver** indicates to DECAF the name of the agent to receive the message.

4. **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.

5. **in-reply-to** is the above reply-with string from a prior message. It is appropriate to think of this and the above field as specifying and using a token which aides 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. **In initiating tasks, in-reply-to must be "null"** or it tells the receiving agent that this (actually original) message is part of an on-going conversation, which the receiving agent will deny.

6. **language** is the language in which the message is expressed. This is usually "DECAF".

7. **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. See Naming, above.

8. **content** is the application specific information to be conveyed by the message. This is domain dependent, but usually follows the $<: \ keyword, value \ >$ pairing conventions of DECAF so that the standard DECAF tools can be used 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 in reply to anything
  K.addFieldValuePair("language", "DECAF"); // always 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 attempt to 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 performatives, Java prohibits them, thus requiring this conversion.

5 Java Code for DECAF Agents

Here we offer little advice – we do not know what you want your agents to do. The code for the SDBW agent (the original is with the demonstration) serves as a partial example of the code needed. This code includes manipulations of the GUI within SDBW. The code below is not an exact copy of the demonstration code.

Observe the following in the Java code below:

1. 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).

2. To understand the contents of the advertising messages, refer to [7].

3. The return from "formatAdv" is "OK" (upper case) while one of the returns from "Response" is "ok" (lower case). DECAF is case sensitive.

4. 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;
    LinkedListQ P;
    Agent local;

    public sdbwStartup()
    { }

    public ProvisionCell formatAdv(LinkedListQ 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");
        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(),"OK");
    }

    public ProvisionCell Response(LinkedListQ Plist, Agent Local) {

        String message = new String(Util.getValue(Plist, "MESSAGE"));
        KQMLmsg K = new KQMLmsg(message);
        System.out.println(K.getValue("content"));
        if(K.getValue("performative").equals("error") &&
           K.getValue("content").equals(":task error Receiving Agent not found"))
           { try {
                Thread.sleep(10000);
            } catch (Exception e) {}
        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", ":receiver " + Local.getName() +
                            " :performative ask-all :ontology Demo " +
                            ":language DECAF :taskName ask_all " +
                            ":parameters parka :keywords " +
                            "Information Extraction");
        return new ProvisionCell(K.getKQMLString(),"fail");
        }
        else { return new ProvisionCell("SOME STRING HERE","ok");}
    }
}
```

6 DECAF's Agent GUI

DECAF agents can be started using a GUI. This GUI is by-passed when parameters are provided on the command line. Normally, the GUI is not used, however, when debugging, the GUI can provide invaluable information on the messages the agent has sent and received, and upon the general state of the agent. Agents can be started with the GUI by issuing the command

```
java Agent
```

This will produce a screen like Figure 8.

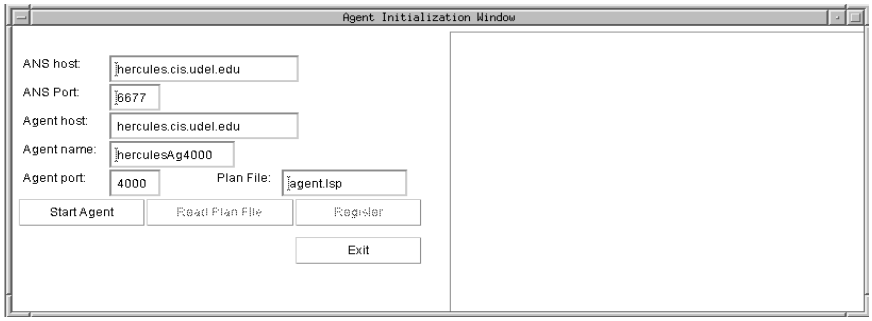

Figure 8: Initial DECAF Agent GUI Screen

The agent now exists, but it is in a useless state. It still needs to know:

- the name of the ANS host with which it should register,
- the agent name under which it should operate, and
- the name of the plan file that describes the agent.

(Other information is beyond the current scope of this documentation, and not described here.)

DECAF will suggest values for all of the needed values.

- For ANS host, it suggests the machine on which it finds itself. Here, it shows "hercules.cis.udel.edu" because that is the name it found.
- For Agent name, it appends "Ag" (for agent) and a port number to the first part of the host name.
- For Plan file, it always suggests agent.lsp

Simply fill in the "Agent name" and provide a source for a "Plan file", press "StartAgent" to read the plan file, and then press "Register" to register the agent and present Figure 9, the base agent screen for the agent.

Since we want to start the Matchmaker, and have already started an ANS on "hercules", we enter "Matchmaker" as the Agent name and "match.lsp" as the Plan file. We can do this from the "arch" directory of the demonstration, because it has copies of the requisite plan file and a jar file containing the needed compiled code. (See [7] for a discussion of the Matchmaker.)

6.1 DECAF Base Agent Screen

The command bar in Figure 9 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,

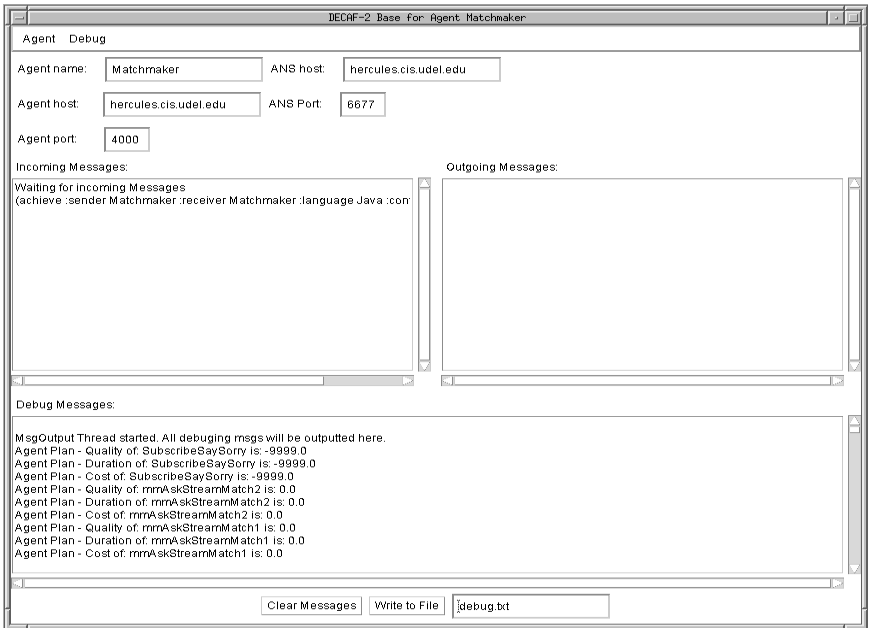

Figure 9: DECAF Base Agent Screen

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 is interesting because it reveals in the debug messages the tasks and actions that have been scheduled for receiving execution time. If, over the course of executing your agent you find it slowing down, it may be that you are inadvertently creating tasks that never get completed – they take up space and need to be checked to see if they can be run. (See Section 3.3.)

Also, an agent with "Size of Tasks: 0" at the bottom of the screen is simply waiting for a message. If that is not your intention, then some activating message has not been sent to the agent. Note that the 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 **it shuts down the whole agent**. Use this option when you are done with the agent, and it 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 you must kill it with Cntl-C or with a system command. If you kill the agent (i.e., the Exit does not work), you may need to use ANSQuery to remove the name of the agent from ANS.

6.2 DECAF KQML Message Sender

To send messages with the GUI, choose "Agent" and then "KQML Message Sender" to get the message sending screen Figure 10. This screen lets you send messages "manually" to agents simply by filling in the needed parts

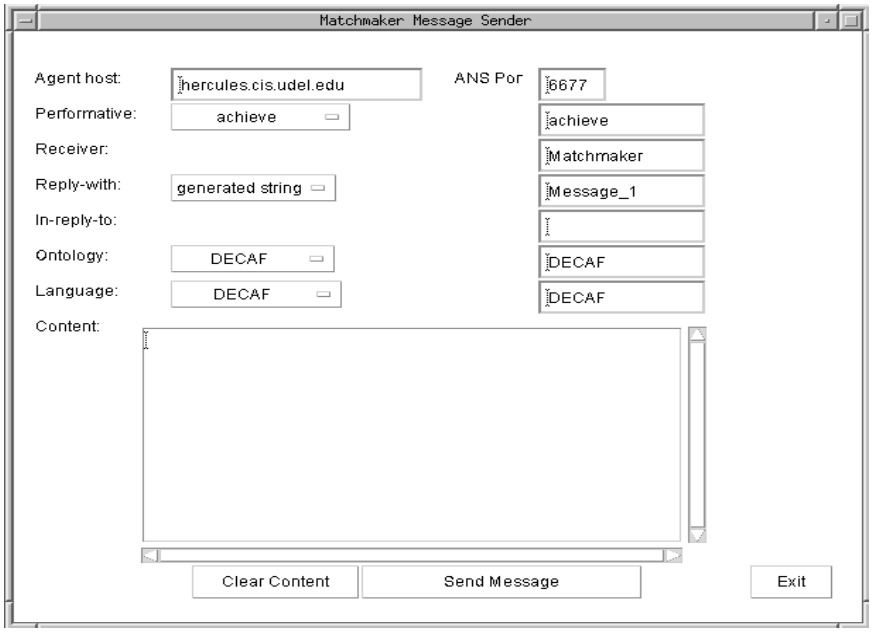

Figure 10: Agent Messaging Screen

and pressing the "Send Message" button.

If you send the message correctly, it will appear 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 will appear 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 9 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 receiving agent can compose a task name.
- **Content** with information for the receiving agent (and which cannot be empty).
- **Language** must be DECAF or Java in most situations.

And here are some hints:

- The "Exit" button on the Message Sender returns you to the base screen.
- To send a Shutdown message, the Ontology field must be empty. DECAF regards _Shutdown as special.
- If you are simulating an agent, you may need to fill in the in-reply-to field, as well.

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.

7 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 that are 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 as public, what they return, and their name.

7.1 Agent Methods

Consider the following code snippet:

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

which is culled from the code examples above. It resembles every method that DECAF invokes because it shows the two arguments passed to the method: a LinkedListQ and an Agent structure. Messages are stored in the LinkedListQ and agent information in the Agent structure. This section deals with methods in the Agent structure. The next section after deals with KQMLmsg structure methods.

public 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 is was invoked.

public void send(KQMLmsg k)

Normally this method is *not* used, as the "cloud" construct is the preferred procedure to make 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.

public void DebugAgent(String Message)

Puts the indicated Message onto the output screen of the base agent GUI screen. (See Section 6.1.) 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.

7.2 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 OriginalReceiver =  Received.getValue("receiver");
  String OriginalSender   =  Received.getValue("sender");
  String OriginalReplyWith=  Received.getValue("reply-with");
  String category  =  Received.getContentFieldValue("category");
  String budget  =    Received.getContentFieldValue("budget");
  if(category.equals("")) category = new String("CategoryMissing");
  if(budget  .equals("")) budget   = new String("0.0");
  . . .
  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. Note that messages are accessed via a utility function Util.getValue (see Section 7.3) 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 3, the term "parka" is used, and the message is also accessible as "parka".

public 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 generally considered as 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., as for "sender" and not ":sender"), and only request one of the eight fields that constitute a KQML message (See Section 4).

public 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 also generally considered as 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 send 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".

public void 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.

public 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");
```

7.3 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.

public static String getValue (LinkedListQ List, String Name)

Returns as a string the contents of the LinkedListQ that has the indicated name. Use of this feature is in general limited to extracting the source message for a task. The code snippet discussed under KQMLmsg Methods (Section 7.2) contains an example of its use to retrieve the MESSAGE that triggered the Task.

public static 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 below to demonstrate the use of the feature. The message to the Matchmaker will timeout after 123 seconds. If you don't particularly care whether a message gets where it's supposed to go, use a timeout of zero, and there will be no waiting for a response.

The Agent has attached to it a Java hashtable for the user to save persistent information. It is defined in the Agent class with the statement:

```
public static Hashtable userHash = new Hashtable ();
```

This Hashtable can be used to store all the objects that you want the agent to have access to. See your documentation on Java to review how this class works. Section 5 contains a one line example of its use (where the query answer is stored in SDBW).

It is sometimes helpful to have Hashtables in Tasks, and these should be declared in a manner similar to that shown above. Note that such tables are only available with the class in which they are declared, and are useful for holding information across several instantiations of the class, which is very helpful when dealing with multiple instantiations of the class or its methods.

8 Correspondence Between Plans and Code

This section is another example of an agent. First, we draw the agent and explain the messages that it requires, and explains the messages that it sends. Second, we show the code that performs the work. Third, we show the correspondence between the plan and the code.

8.1 A Random Number Generator

Figure 11 shows the plan file for a rather unusual random number generating agent. This agent requires a message that states whether the random number to be returned is to be even or odd. The agent itself generates

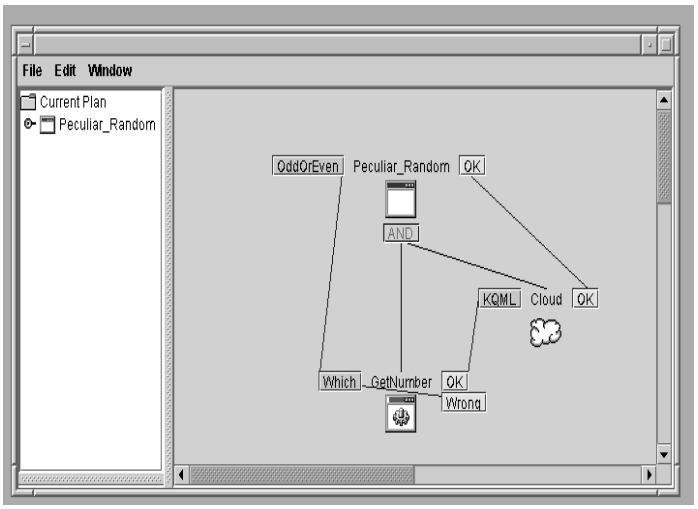

Figure 11: Planeditor Plan for Peculiar_Random

random numbers and tests them. The parity of a generated number is correct if the generated number is odd number when an odd number is requested or is an even number when an even number is requested. If the parity is wrong, the agent exits by way of the "Wrong" outcome, and sends the original specification back to itself for another try. If the parity is correct, then the agent exits the GetNumber action by way of the "OK" outcome, returning the random number and the number of tries it took to generated the number (tries should, of course, follow a binomial distribution).

The message to initiate this agent must

- know the name of the agent, and the code below does not advertise,
- set **receiver** to the name of the agent, *which must be known somehow* (see above)
- set **sender** to the name of the sender, *which the sender would know*
- set **performative** to *achieve*
- set **ontology** to *Peculiar*
- set **language** to *DECAF*
- set **reply-with** to *null* or to *some value*
- set **in-reply-to** to *null*, or this agent will be confused, and will not be enabled to generate a random number
- set **content** to include *:task Random* and *:OddOrEven even* to get an even number or *:OddOrEven any-value-except-even* to get an odd number in return.

The performative is "achieve" because this random number generator is not named for a standard communicative act. For this reason, it is necessary to include the ":task Random" in the content field. The concatenation of the ontology with the task name produces the name of the Java class: "Peculiar_Random".

"OddOrEven" is a provision that must be satisfied in the incoming message for the Task to be instantiated. A value of "even" gets an even random number, any other value produces an odd random number. The random numbers range over the value of integers in Java, and may be positive or negative (which is why we square the modulo result).

The agent returns two values. The first has the keyword "result" and is the random number of the proper parity. The second has the keyword "tries" to state the number of tries it took to generate a random number of the proper parity.

8.2 Code for the Agent

One possible Java class for the above agent is shown below. This code is not the most efficient possible, but permits the following observations:

- "MESSAGE" is the envelope (in the sense that it contains all eight message fields from the sender) for the request for a random number. This need not be processed until after the number is found. That is, it is only examined once in the above code, not for each invocation of GetNumber.
- The KQMLmsg K is instantiated as an exact copy of the incoming message M. Since all eight values are set, the manipulation of K need only change those that need to have different values in the return message. Usually, the ontology and the language do not change. Also, it is usual to swap the sender and receiver, to copy the "reply-with" field to the "in-reply-to" field, and to set the reply-with field only if a reply is expected.
- The performative is set to "tell" in the return message. This is a "safe" thing to do because we have not examined the reply-with field sent in the incoming message M. We do not dictate how the sender will handle the response, but the reply-with field indicates the senders intention:
  - If reply-with was null, the sender wants a "tell" message, and the code provides that. The sender will have a Task named Peculiar_tell to accept the response.
  - If reply-with was non-null, the sender wants the response to be a reply, in which case the performative is ignored, and this code handles that situation as well. (A "tell" can be ignored as easily as "achieve".) The sending agent is waiting for a response from the cloud in which this Task is located.
- The return string for the OK outcome is constructed with a timeout, but the timeout is in fact redundant – the null reply-with field already indicates that no reply is expected.

- The string supplied in the outcome labeled “Wrong” becomes the string that is the value of the variable “Which” on entry to the method GetNumber. To preserve this value, it is necessary to send it out as the first value in the returned ProvisionCell.
- The coding has been made more confusing than necessary because two different names are used for the “OddOrEven” variable which drives the whole calculation. It should have been left as “OddOrEven” for the entire agent, and certainly within a class. The line from “OddOrEven” to “Which” should not rename the provisions or parameters as a matter of good programming style. The GetNumber method does not have access to the value of OddOrEven except through the content field of the message named ”MESSAGE”.
- This coding can fail to return proper values for the “tries” variable when it is invoked concurrently. This is a use for hash tables, and that use is left as an exercise. See Section 8.4.

Herewith, the code . . .

```
import java.io.*;
import java.util.*;

public class Peculiar_Random
{
   static Random Randy = new Random();
   int tries;

   public Peculiar_Random()
   {
      tries = 0;
   }

public ProvisionCell GetNumber(LinkedListQ Plist, Agent Local)
 {
  String Which = new String(Util.getValue(Plist, "Which"));
  int rand = Randy.nextInt();
  tries++;
  int test_value = rand % 2;
  test_value = test_value * test_value;
  int want_value = 1;
  if(Which.equals("even")) want_value = 0;
  if(test_value == want_value) // correct parity
   {
     KQMLmsg M = new KQMLmsg(Util.getValue(Plist, "MESSAGE"));
     KQMLmsg K = new KQMLmsg(M);
     K.addFieldValuePair("performative", "tell");
     K.addFieldValuePair("sender", Local.getName());
     K.addFieldValuePair("receiver", M.getValue("sender"));
     K.addFieldValuePair("in-reply-to", M.getValue("reply-with"));
     K.addFieldValuePair("reply-with", "");
     K.addFieldValuePair("content",
              ":result "+rand+" :tries "+tries);
     String X = K.getKQMLString();
     X += Util.addTimeout(0);
     return new ProvisionCell( X, "OK");
   }
   else // not correct parity
   {
     return new ProvisionCell( Which, "Wrong");
   }
 }   // to close GetNumber
} // to close the Peculiar_Random Class
```

We analyze the above code in the next section.

8.3 Diagram of Plan and Code

The WeiGram[1] for the Peculiar_Random class is shown in Figure 12. A WeiGram divides the code for an agent into the same number of items as appear in the plan file for an agent. Since each item has Java code to execute its action the division provides non-empty boxes for each Action and each Task that reduces to Actions. Clouds

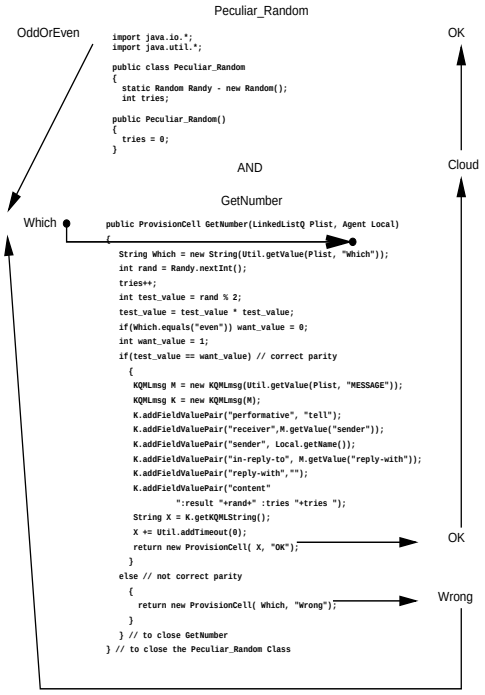

Figure 12: WeiGram for Peculiar_Random

contain no code in WeiGrams, as the user does not code them. The top-most method in each box in a WeiGram is the Java method that is invoked by DECAF when the Action is implemented. Utility routines for the Agent, such as its own private functions can be put in any box, but if used by only one method are best put in the box for the using method.

Arrows on the WeiGram trace the value of the provision "OddOrEven" to "Which" (see above discussion on why the name change is a bade idea), shows the correspondence between the provision and the variable in the Java code, and shows how the value of "Which" is returned to the next invocation of GetNumber. Lines strictly outside the boxes correspond exactly to lines on Figure 11. The lines from the return statements in GetNumber are drawn to the outcomes that they provide.

Finally, note that it is the cloud, not the agent, that sends the response to the invoking agent. Without the cloud, the message that GetNumber forms would disappear. DECAF does not send messages unless directed to by a plan file that directs the outcome string to a cloud. If Figure 11 did not have a cloud between the OK outcome of GetNumber and the outcome OK of the Task Peculiar_Random, the message that GetNumber forms would disappear. Further, if the outcome from GetNumber were directed to *both* the outcome OK of the Task Peculiar_Random and the cloud, there would be a "race condition" as to whether DECAF removed the instantiation of the Peculiar_Random Task before or after the cloud completed. If the instantiation of the Peculiar_Random Task is removed first, the message never gets sent.

[1]This is a diagram of a form inspired by Wei Chen of the Computer and Information Science Department.

8.4 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.

9 Conclusion

This has been a short introduction to DECAF programming. DECAF is a multi-agent system building toolkit that provides (1) a Plan-Editor GUI to easily establish the control mechanisms needed to express exchanges ("conversations") between participants (agents) and (2) the operating system features to perform the message sending and receiving that supports domain-level constructs. 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 help researchers and students get to the "interesting" parts of a multi-agent system or simulation more quickly.

Here we have

- further explained the DECAF demonstration
- given example of Programming the Agent
- shown how to edit items in the plan file
- shown the Properties Window
- demonstrated using the DECAF GUI
- briefly explained KQML messages
- given limited advice on writing Agent Action code
- made passing reference to Matchmaker [7] and Broker [6].
    - The Matchmaker agent matches service needs with agents that provide the service or with a Broker that may have several providers of that service
    - The Broker agent can select from several agents that provide the same or similar service based on some quality of service (cost, duration,quality)
- shown where to get additional reference material
- and in general provided a helpful, interesting, well-written document.

This background, together with the demonstration code and other DECAF reference materials, should provide the interested researcher with sufficient material to implement DECAF agents that provide useful services.

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 Graham, Michael Mersic, and Keith Decker. Scheduling and scalability in multi agent systems. Technical report, University of Delaware, Department of Computer and Information Science, May 2000. Technical Report 2000-02.

[3] John Graham, Michael Mersic, and Keith Decker. Support for research management in multi agent systems. Technical report, University of Delaware, Department of Computer and Information Science, May 2000. Technical Report 2000-03.

[4] John R. Graham, Michael Mersic, and Keith S. Decker. Scalability and scheduling in an agent architecture. Technical Report TR-2000-03, University of Delaware, 2000.

[5] John R. Graham, Michael Mersic, and Keith S. Decker. Support for resource management in multi-agent systems. Technical Report TR-2000-02, University of Delaware, 2000.

[6] Mikko Laukkanen and Jukka Eskelinen. Requirement specification for the decaf-broker. Technical report, University of Delaware, May 1999.

[7] Mikko Laukkanen and Jukka Eskelinen. Requirement Specification for the DECAF-Matchmaker. Technical report, University of Delaware, May 1999. Updated June 2000 by Foster McGeary.

[8] Foster McGeary and Keith Decker. Modeling a Virtual Food Court Using DECAF. In *MABS 2000, The Second Workshop on Multi-Agent Based Simulation at ICMAS 2000, The Fourth International Conference on MultiAgent Systems*, Boston MA, USA, July 2000.

[9] A.S. Rao and M.P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 312–319, San Francisco, June 1995. AAAI Press.

[10] Stuart Russell and Eric Wefald. *Do the Right Thing: Studies in Limited Rationality*. The MIT Press, Cambridge, Massachusetts, 1991.

[11] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J O'Sullivan. A layered architecture for office delivery robots. In *Proceedings of the 1st Intl. Conf. on Autonomous Agents*, pages 245–252, Marina del Rey, February 1997.

[12] 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.