Towards a Distributed, Environment-Centered Agent Framework

John R. Graham and Keith S. Decker
Department of Computer and Information Sciences
University of Delaware
{graham, decker}@cis.udel.edu

Abstract

This paper will discuss the internal architecture for an agent framework called DECAF (Distributed Environment Centered Agent Framework). DECAF is a software toolkit for the rapid design, development, and execution of "intelligent" agents to achieve solutions in complex software systems. From a research community perspective, DECAF provides a modular platform for evaluating and disseminating results in agent architectures, including communication, planning, scheduling, execution monitoring, coordination, diagnosis, and learning. From a user/programmer perspective, DECAF distinguishes itself by removing the focus from the underlying components of agent building such as socket creation, message formatting, and agent communication. Instead, users may quickly prototype agent systems by focusing on the domain-specific parts of the problem via a graphical plan editor, reusable generic behaviors [3], and various supporting middle-agents [4]. This paper will briefly describe the key portions of the DECAF toolkit and as well as some of the internal details of the agent execution framework. While not all of the modules have yet been completely realized, DECAF has already been used for teaching purposes, allowing student teams, initially untutored in agent systems, to quickly build prototype multi-agent information gathering systems.

Keywords: Agent-based **SOFTWARE ENGINEERING**, Intelligent agents, Internal agent architecture, Action scheduling, Multi-agent systems.

1 Introduction

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 execute intelligent agents to achieve solutions in complex software systems. DECAF provides the necessary architectural services of a large-grained intelligent agent [6, 20]: communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis [14]. This is essentially, the internal "operating system" of a software agent, to which application programmers have strictly limited access.

The control or programming of DECAF agents is provided via a GUI called the *Plan-Editor*. In the Plan-Editor, executable actions are treated as basic building blocks which can be chained together to achieve a larger more complex goal in the style of an HTN (hierarchical task network). This provides a software component-style programming interface with desirable properties such as 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 [23, 5].

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 UMASS [9, 21].

The goals of the architecture are to develop a modular platform suitable for our 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 [4] 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. Conceptually, we think of DECAF as an agent operating system. Code within that action can make calls to the DECAF framework to send messages, search for other agents or implement a formally specified coordination protocol. This interface to the framework is a strictly 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 knowledgebase of the framework.

This paradigm differs from some well known toolkits which use the API approach to agent construction, three of which are mentioned here. *JAFMAS* (JAVA based Agent Framework for Multi-Agent Systems) at the University of Cinncinati provides a set of JAVA classes that must be extended and therefore the programmer must understand the underlying interfaces. In particular, the programmer must explicitly establish each conversation, rules for the particular conversation and startup procedures through the specified JAVA interface.

Bond, is a Java-based, object-oriented middleware for network computing currently under development at Purdue University. In bond, the focus of the programming paradigm is the network and the message passing interface. *Middleware* is a software layer that allows developers to mold systems tailored to specific needs from components and develop new components based upon existing ones. The basic components of Bond are meta-objects and agents. Meta-objects provide synthetic information about network resources and agents use this information to access and manipulate network objects on behalf of users.

JATLite from Stanford University, is a set of Java packages that make it easy to build multi-agent systems using Java. JATLite provides a basic infrastructure in which agents register with an Agent Message Router facilitator using a name and password, connect/disconnect from the Internet, send and receive messages, transfer files, and invoke other programs or actions on the various computers where they are running. The basic programming component with JATLite is the JAVA API which the programmer will use to build all the functionality needed by a network based agent.

In contrast, DECAF allows the agent developer to immediately program at the *whole-agent* level—i.e. basic domain actions and desired capabilities—very rapidly without the need to focus on the messages or the network of any JAVA specific classes. By providing a GUI plan interface for agent programming the user can immediately begin describing capabilities and programming the agent actions. This paper will briefly describe the key portions of the DECAF toolkit and as well as some of the internal details of the agent execution framework.

2 DECAF Operation

The basic operation of DECAF requires three components:

- An Agent Name Server (ANS)
- An Agent Program or Plan File
- The DECAF Framework

The purpose of the ANS is similar to most name servers such as DNS (Domain Name Server) or the portmapper on generic UNIX systems. The idea is that a new agent will register its existence with the ANS. Such registration consists of a socket number and a host name. Once the agent is registered any other agents wishing to communicate will first contact the ANS to determine if the recipient is currently up and working. If so, the ANS will respond with the address and further communications will be carried on directly between agents. This is like looking up someone's name in the phone book (white pages) and then making the call. If the name is not in the phone book you will not be able to make the call. Similarly, if your agent has not registered with the ANS, no other agent will be able to communicate with it. Agent and ANS interaction a shown in figure 1. From the agent programmer's perspective, the interactions with the ANS occur automatically and behind the scenes. This capability is fairly routine among implemented agent systems. DECAF currently uses the CMU RETSINA ANS.

The plan file is the output of the Plan Editor and represents the programming of the agent. One agent consists of a set of capabilities (potential objectives, goals, or desires in BDI nomenclature) and a collection of actions that may be planned and executed to achieve the objective. These capabilities can correspond to classical AI black-and-white goals or "worth-oriented" objective functions over states [16, 22]. Currently, each capability is represented as a complete task reduction tree (HTN [7]), similar to that in RETSINA [22], with annotations drawn from the TÆMS task structure description language [5, 21]. The leaves of the tree represent basic agent actions (HTN primitive tasks). One agent can have dozens or even hundreds of actions. The basic actions must be programmed in a precise way just as any program written in C or Java must be. However, the expression of a plan for providing a complete capability is achieved via program building blocks that are not simple declarative programming language statements but are a sequence of actions connected in a manner that will achieve the goal[1]. Actions are reusable in any sequence for the achievement of many goals. Each of these capabilities and the programming to achieve the associated goal is specified in a *Plan file*.

In our toolkit, a Plan file is created using a GUI called the *Plan-Editor*. This interface was influenced by work such as the software component editor for ABE [13] and the TÆMS task structure

[1]This should come to no surprise to people familiar with HTN planning, but is a small conceptual hurdle for non-AI-trained agent programmers

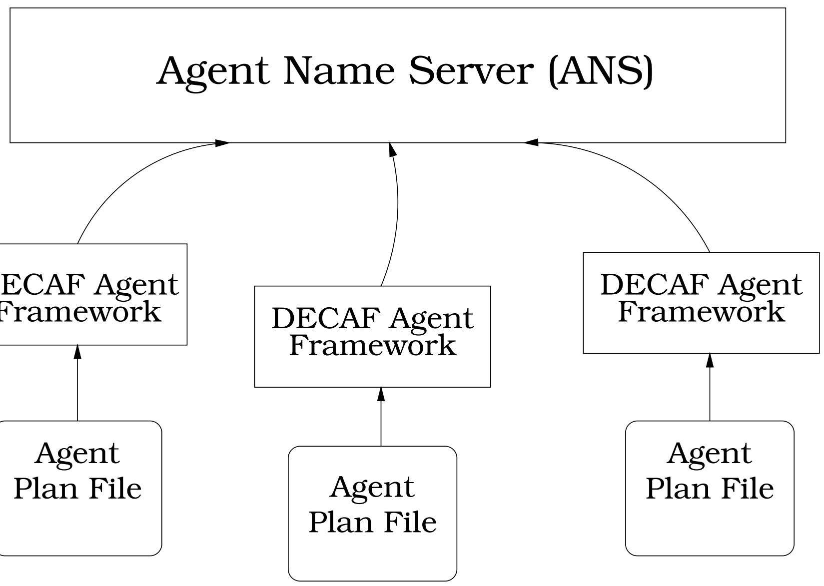

Figure 1: ANS , Agent Interaction

editor for the Boeing MADEsmart/RaDEO project [1]. In the Plan-Editor, a capability is developed using a Hierarchical Task Network-like tree structure in which the root node expresses the entry point of this capability "program" and the goal to be achieved. Non-leaf nodes (other than the root) represent intermediate goals or compound tasks that must be achieved before the overall goal is complete. Leaf nodes of the tree represent actions. Each task node (root, non-root and leaves) has a set of zero or more inputs called *provisions* , and a set of zero or more *outcomes* [23]. The provisions to a node may come from different actions, so no action will start until all of its provisions have been supplied by an outcome being forwarded from another node (this may of course be an external node, e.g. a KQML or other ACL message). Provision arcs between nodes represent the most common type of inter-task constraint (they are a subclass of the TÆMS *enablement* relationship) on the plan specification.

A node, N_i, may have multiple outcomes but the node is considered complete as soon as one outcome has been provided. The *outcomes* represent a complete *classification* or partition of the possible results. For example, the outcome of node N_i may be any outcome in the set of outcomes O_1, O_2,. . . , O_n. If the result of the action is classified as outcome O_k then control will pass to node N_k, if the result is classified as outcome O_l control will pass to node N_l. In either case, as soon as an outcome is provided that node has completed processing. In this way conventional looping or conditional selection can be created by specifying correct control. The Plan-Editor allows such control to be specified simply by drawing boxes and connecting the outcomes to other boxes. When the picture is drawn, saving the file will create an ASCII plan file that is used to program the actions of the agent. A picture of a sample Plan-Editor session is shown in Figure 2.

In Figure 2, *Sample* is the name of the goal, *urlSearch*, *Search1*,*Search2* and *A1* are the actions. When the provision for Sample, named *stock*, is provided, the urlSearch action will execute and depending on the outcome (*ok* or *fail*) either Search1 or A1 will execute followed by Search2. When Search2 is complete, the outcome (*result*) is provided to the goal and the goal has been completed.

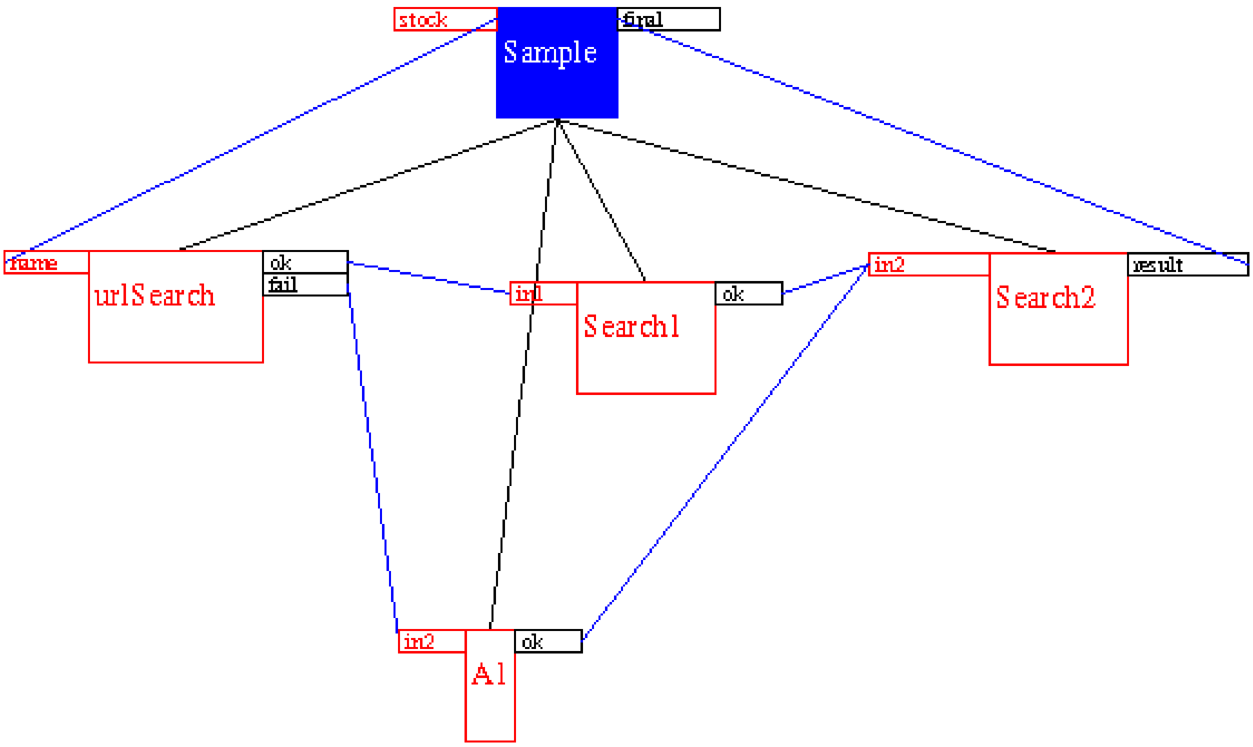

Figure 2: Sample Plan File

3 DECAF Implementation

Figure 3 represents the high level structure of the DECAF architecture. Structures inside the heavy black line are internal to the architecture and the items outside the line are user-written or provided from some other outside source (such as incoming KQML messages).

As shown in Figure 3, there are five internal execution modules (square boxes) in the current DECAF implementation, and seven associated data structure queues (rounded boxes).

3.1 Execution modules

3.1.1 Agent Initialization

The execution modules control the flow of a task through its life time. After initialization, each module runs continuously and concurrently in its own Java thread. When an agent is started, the agent initialization module will run. The *Agent Initialization* module will read the plan file as described above. Each task reduction specified in the plan file will be added to the *Task Templates Hashtable* (plan library) along with the tree structure that is used to specify actions that accomplish that goal.

Note that the initializer must be concerned with several features of the Plan File. First there may be directives, similar to C preprocessor directives that indicate external files must be read (Similar to the *import* directive in Java). These files will be read and incorporated into the plan templates.

Next the plan may make use of a *Startup* module. The Startup task of an agent might, for example, build any domain or beliefs data/knowledgebase needed for future execution of the agent. Any

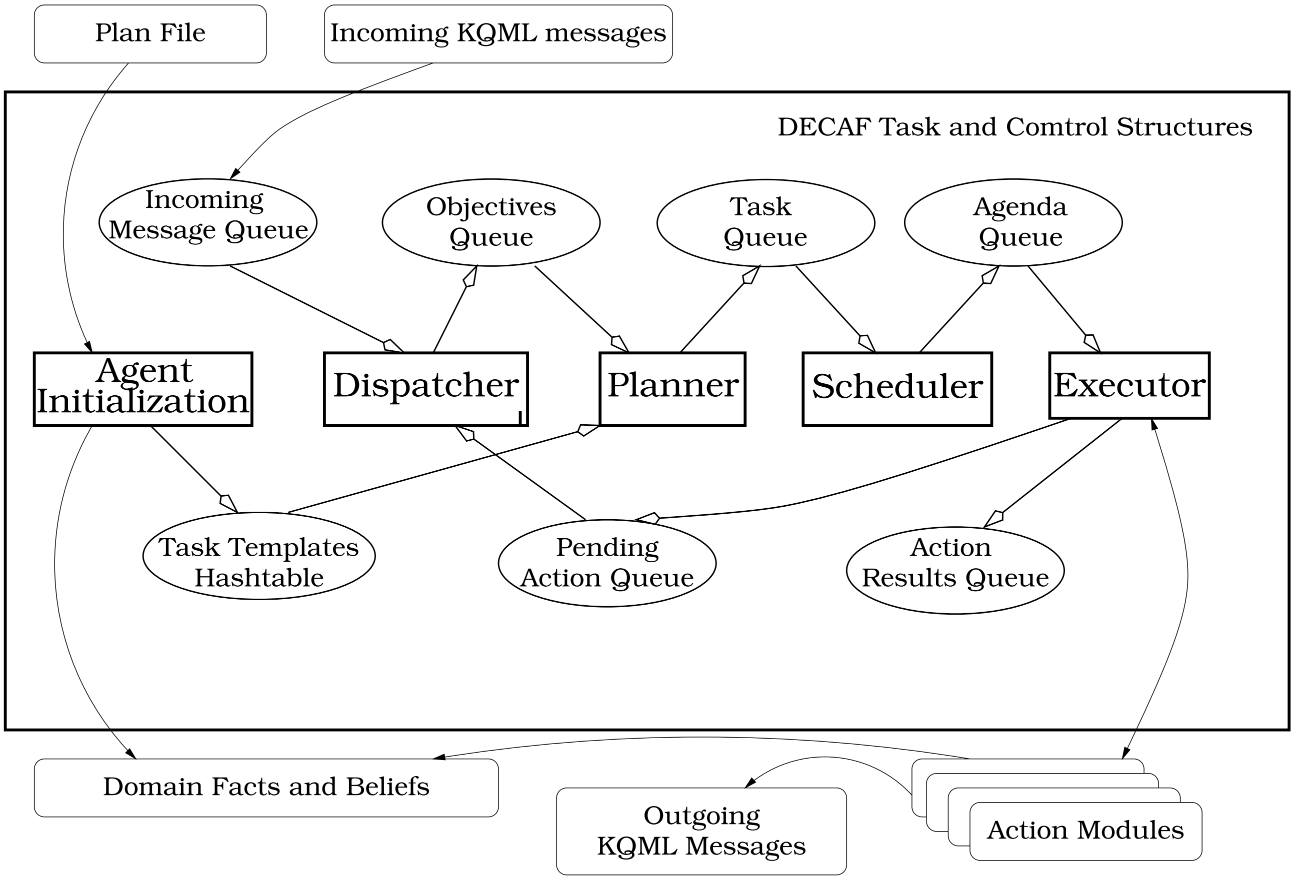

Figure 3: DECAF Architecture Overview

subsequent changes to initial data must come from the agent actions during the completion of goals. Startup tasks may assert certain continuous maintenance goals or initial achievement goals for the agent. Actions are normally initiated as a result of an incoming message requesting that action. The Startup task is special since no message will be received to begin its execution. If such a Startup module is part of the plan file, the initialization module will add it to the *Task Queue* for immediate execution.

Lastly, the plan file may be incomplete in the sense that some portions of the plan will not be known until the results of previous actions are complete (interleaved planning and execution). In this case the initialization module will build place holders in order to complete the action tree.

Specific task structures are read from the plan file, are listed in the plan library, and are not currently changed during the agent lifetime (but see the discussion of the DECAF planner). The last thing the Agent Initialization Module does is register with the ANS and set up all socket and network communication.

3.1.2 Dispatcher

Agent initialization is done once and then control is passed to the Dispatcher which waits for incoming KQML messages which will be placed on the *Incoming Message Queue.* An incoming message contains a KQML *performative* and its associated information. An incoming message can result in one of three actions by the dispatcher. First the message is attempting to communicate as part of an ongoing conversation. The Dispatcher makes this distinction mostly by recognizing the KQML `:in-reply-to` field designator, which indicates the message is part of an existing conversation. In this case the dispatcher will find the corresponding action in the *Pending Action Queue* and set up the tasks to continue the agent action.

Second, a message may indicate that it is part of a new conversation. This will be the case whenever the message does not use the `:in-reply-to` field. If so a new *objective* is created (equivalent to the BDI "desires" concept[15]) and placed on the *Objectives Queue* for the Planner. An agent typically has many active objectives, not all of which may be achievable. The last thing the Dispatcher is responsible for is the handling of error messages. If an incoming message is improperly formatted or if another internal module needs to sends an error message the Dispatcher s responsible for formatting and send the message.

3.1.3 Planner

The Planner monitors the Objectives Queue and matches new goals to an existing task template as 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. If a subsequent message comes in requesting the same goal be accomplished, then another instantiation of the same plan template will be placed in the task networks with a new unique identifier. 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 an incoming request.

3.1.4 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. This determination is currently based on whether all of the provisions for a particular module are available. Some provisions come from the incoming message and some provisions come as a result of other actions being completed. This means the Tasks Queue Structures are checked any time a provision becomes available to see which actions can be executed now.

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 [15]—that of balancing reconsideration of activities in a dynamic, real-time environment with taking action [17]. Rather than taking the formal BDI model literally, we develop the deliberative components based on the practical work on robotics models, where the so called "3T" (three tier) models have proven extremely useful [8, 19, 2]: here, Planning, Scheduling, and Execution Monitoring. Each level has a much tighter focu, 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 first implementation of the Scheduler, the order was first-come-first-served (FCFS). As any basic Operating Systems student knows, this can result in some very long wait times for tasks. On the other hand, determining an optimal schedule for a set of tasks in a fixed period of time can be an NP-hard problem. In the research section of this paper some heuristics for determining schedule are described (see Section 5.2)

3.1.5 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 into execution. One of two things can occur at this point: The action can complete normally. (Note that "normal" completion may be returning an error or any other 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.

The other case is when the action partially completes and returns with an indication that further actions will take place later. This is a typical result when an action sends a message to another agent requesting information, but could also happen for other blocking reasons (i.e. user or internet I/O). The remainder of the task will be completed when the resulting KQML message is returned. To indicate that this task will complete later it is placed on the *Pending Action Queue*. Actions on this queue are keyed with a *reply-to* field in the outgoing KQML message. When an incoming message arrives, the Dispatcher will check to see if an *in-reply-to* field exists. If so, the Dispatcher will check the Pending action queue for a corresponding message. If one exists, that action will be returned to the Agenda queue for completion. If no such action exists on the Pending action queue, an error message is returned to the sender.

4 Agent Construction

One of the major goals of building the DECAF framework is to enable very rapid development of agent actions and agent programming. This is accomplished by removing the agent interaction requirement from the programmers hands. The developer does not need to write any communications code, does not have to worry about multiple invocations of the same agent, does not have to parse and schedule any incoming messages, and does not have to learn any Application Programmer Interface (API) in order to write and program an agent under the DECAF architecture. Note that since the Plan File incorporates all of the data flow of an agent, the programmer does not have to write code for data flow between actions either.

The plan file represents the agent programming and each leaf node of the program represents a program that the user must write. Each action takes as inputs (or parameters) a list of provisions and produces as output a result that can be classified as exactly one *outcome*. When writing agent code, there will be one (Java) method corresponding to the entry point of the agent action. This entry method consists of a constructor and a method named *StandardMethod*. This method takes as arguments a linked list of the input parameters and the Agent class. Utilities are provided for extracting the parameters and the Agent handle is used to send out KQML messages.

The method (and any agent method) returns a *Provision Cell*. The contents of a Provision Cell are a provision name and a value for that provision. The provision name will be set to one of the outcome provision names specified in the Agent plan file. A special provision name, *Pending* is used to indicate that this action is not complete yet, and will be receiving further messages as part of an ongoing conversation. When the Pending keyword is used as a name, the Provision Cell value is set to the name of the method that should be entered when the response message arrives. Using this mechanism, a finite state machine can be setup by reentering your code based on specific results.

Figure 4 shows an agent action named *Search4*. The code shows a template for an action that starts, sends a KQML message and when the response to the message arrives, the action is reentered in the *ResultMethod* method of the code.

5 Current Activities and Research

The DECAF architecture is particularly well suited as a research and development platform. Since each module is a separate thread, one area can be singled out for research without impact on other activities. One research project is centered around performance of both the framework and the performance of agents.

5.1 Software Engineering Perspectives

One of the major goals of building the DECAF architecture was to enable people without any agent development experience to quickly and easily develop and program agents that would interact to achieve a larger goal. This concept was verified by using the framework and tools in a recent class on Information Gathering. The class consisted of fifteen students who had never done any agent programming and most of whom had never done Java programming (although all had C or C++ experience). Two lectures were presented, one on the basic concepts of agents programming and one on the specifics of agent writing in our framework. In the following four weeks, approximately twenty agents were developed using around forty separate agent actions programmed to accomplish objectives. The agents were combined into two prototype multi-agent systems, one to do Warren-like financial portfolio management[20], and one to evaluate and track graduate school applications.

In addition to debugging and proving the stability of the DECAF framework, this experience provided the desired development platform for agent development and programming. One of the major purposes of the project is to develop a tool for teaching agent systems concpets; to this end we will be making DECAF publically available this summer.

5.2 Schedule Design Criteria

One problem associated with large scale agent development is applying real-time performance constraints to the system. The notion of an *Anytime Agent* [18] is used to allow agents to search until a specific criteria or quality of answer is achieved. Real-time performance may also be achieved by conditional scheduling of agents using sampling [11]: An initial schedule is laid out and then may be changed based on the sampling results. A third method for achieving real-time results is known as *Design to Time*. [10] The idea to specify a deadline when an agent must be completed. The agent will then determine the method to achieve an optimal result in the time given.

In general, determining an optimal schedule of agents, even a group of agents with predetermined performance characteristics is at least an NP-hard problem. The modularity of DECAF provides a particularly good base from which to investigate various heuristics for scheduling possibilities.

```
// Libraries to be imported.
import java.io.*;
import java.net.*;
import EDU.cmu.softagents.misc.ANS.ANSClient.*;
import EDU.cmu.softagents.util.*;
import EDU.cmu.softagents.misc.KQMLParser.*;

public class Search4 {

    public Search4()  // the zero argument constructor for testing
    {
        System.out.println(" The Zero constructor");
    }
    public ProvisionCell StandardMethod(LinkedListQ Plist, Agent Local)
    {
        ProvisionCell Result = new ProvisionCell();
        String Pvalue = new String();
        String Pname = new String();
        String Input = new String();

        // build the message
        String Message = Util.getValue(Plist,"MESSAGE");
        KQMLmessage K = new KQMLmessage(Message);

        // Action logic goes here
        // build a properly formed KQML
        // message for sending if needed
        // Use the framework for sending
        Local.send(OutgoingMessage);


        // Set up the return results
        Pname = "pending";
        Pvalue = "ResultMethod";
        Result = new ProvisionCell(Pvalue,Pname,null);
        return (Result);

    }  // End of StandardMethod

    public ProvisionCell ResultMethod(LinkedListQ Plist, Agent Local)
    {
        ProvisionCell Result = new ProvisionCell();
        String Pvalue = new String();
        String Pname = new String();
        String Input = new String();

        // build the message and send if needed
        String Message = Util.getValue(Plist,"MESSAGE");
        Local.send(OutgoingMessage);

        Pname = "ok";
        Pvalue = "This is a very long string";
        Result = new ProvisionCell(Pvalue,Pname,null);
        return (Result);

    }  // End of ResultMethod

}
```

Figure 4: Sample Agent Code

5.3 Threading

The entire Framework makes heavy use of Java Threads. Each execution module is a separate thread when the framework is started. Currently each task run by the Executor is run sequentially. Research is being done to determine if each task run by the Executor should be a separate thread. On one hand this will prevent any undue delay by tasks waiting to run since each task will gets its own thread an run immediately. Control would then be immediately returned to Executor for selection of the next task. The negative aspect of this design is data synchronization. This is easily enough solved but will complicate the programming interface significantly. One possible solution to this would be to have the agent programmer specify if this action is capable of independent activity by setting a flag in the performance profile.

Another drawback to accomplishing thread synchronization also is the lack of synchronization primitives in the Java language. Java uses *Monitors* as the sole means of process/thread synchronization. The DECAF Framework has included in it for internal use: binary semaphores, counting semaphores, condition variables and reader/writer locks. The basis of this language enhancement comes from [12].

Impact on performance is also a consideration. The internals of the Java Virtual Machine (JVM) are not clear when running on multiple processors and the future implementation of the JVM may change.

6 Acknowledgments

The initial Framework was built by Vasil Hynatishin, and the Plan-editor by David Cleaver. Internal modification to the Framework structure was done by John Graham and upgrades to the Agent GUI was done by Anthony Rispoli. Terry Harvey also contributed to proofreading this paper. This project is supported by the National Science Foundation under IIS-9812764 and IIS-9733004.

References

[1] T. Barrett, G. Coen, J Hirsh, L Obrst, J Spering, and A. Trainer. MADEsmart: An integrated design environment. 1997 ASME Design for Manufacturing Symposium, 1997.

[2] R.Peter Bonasso, David Kortenkamp, and Troy Whitney. Using a robot control architecture to aoutomate space shuttle operation. In *Proceding of the Ninth Conference on Innovative Appications of AI*, 1997.

[3] K. S. Decker, A. Pannu, K. Sycara, and M. Williamson. Designing behaviors for information agents. In *Proceedings of the 1st Intl. Conf. on Autonomous Agents*, pages 404–413, Marina del Rey, February 1997.

[4] K. S. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 578–583, Nagoya, Japan, August 1997.

[5] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 217–224, Washington, July 1993.

[6] Keith S. Decker and Katia Sycara. Intelligent adaptive information agents. *Journal of Intelligent Information Systems*, 9(3):239–260, 1997.

[7] K. Erol, D. Nau, and J. Hendler. Semantics for hierarchical task-network planning. Technical report CS-TR-3239, UMIACS-TR-94-31, Computer Science Dept., University of Maryland, 1994.

[8] R. James Firby. Task networks for controlling continuous processes. Seattle, WA, 1996.

[9] Alan Garvey, Marty Humphrey, and Victor Lesser. Task interdependencies in design-to-time real-time scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 580–585, Washington, July 1993.

[10] Alan Garvey and Victor Lesser. Design-to-time real-time scheduling. COINS Technical Report 91–72, University of Massachusetts, 1991. To appear, IEEE Transactions on Systems, Man and Cybernetics, 1993.

[11] Lloyd Greenwald and Thomas Dean. A conditonal schdeuling approach to designing real-time systems. *The Journal of AAAI*, 1998.

[12] Stephen J. Hartley. *Concurrent Programming, The Java Programming Lanaguage*. Oxford University Press, Drexel University, 1998.

[13] F. Hayes-Roth, L. Erman, S. Fouse, J. Lark, and J. Davidson. ABE: A cooperative operating system and development environment. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 457–490. Morgan Kaufmann, 1988.

[14] B. Horling, V. Lesser, R. Vincent, A. Bazzan, and P. Xuan. Diagnosis as an integral part of multi-agent adaptability. Tech Report CS-TR-99-03, UMass, 1999.

[15] 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.

[16] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers*. MIT Press, Cambridge, Mass., 1994.

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

[18] Tuomas Sandholm and V. Lesser. Utility-based termination of anytime agents. CS Technical Report 94–54, Univ. of Massachusetts, 1994.

[19] Reid Simmons. Becoming increasingly reliable. 1996.

[20] K. Sycara, K. S. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6):36–46, December 1996.

[21] 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.

[22] M. Williamson, K. S. Decker, and K. Sycara. Executing decision-theoretic plans in multi-agent environments. In *AAAI Fall Symposium on Plan Execution*, November 1996. AAAI Report FS-96-01.

[23] M. Williamson, K. S. Decker, and K. Sycara. Unified information and control flow in hierarchical task networks. In *Proceedings of the AAAI-96 workshop on Theories of Planning, Action, and Control*, 1996.