

Real-Time Scheduling for Distributed Agents *

John R. Graham

Department of Computer and Information Sciences

Keith S. Decker, Thesis Advisor

University of Delaware

Newark, DE 19716

Abstract

Providing an environment for a software agent to execute is very similar to building an operating system for the execution of general purpose applications. In the same fashion that an operating system provides a set of services for the execution of a user request, an agent framework provides a similar set of services for the execution of agent actions. Such services include the ability to communicate with other agents, maintaining the current state of an executing agent, and selecting an execution path from a set of possible execution paths. The particular focus of this paper is the study of Soft Real-Time agent scheduling in the context of a framework for the execution of intelligent software agents; a characterization of agent performance; and development of an environment for testing and comparing the performance of agent activities. The agent architecture used for this study, DECAF (Distributed Environment Centered Agent Framework), is a software toolkit for the rapid design, development, and execution of “intelligent” agents to achieve solutions in complex software systems. Unlike a traditional operating system, DECAF has the ability to reason about action execution if a characterization of action performance is available. Also featured in DECAF is the ability to reason about deadlines and other commitments. In this sense DECAF supports the idea of “soft” real time execution of tasks. To achieve this, the concept of execution profiles and a characterization of agent execution that will lead to optimal or near optimal scheduling of agent execution is presented. This paper will discuss how an agent architecture differs from an operating system, the essential details of the of the agent execution framework, a formal model of the parameters used for scheduling considerations and how the agent architecture has been enhanced to provide real-time servicing of agent requests.

Keywords: Agent-based **REAL-TIME SYSTEMS**, Intelligent agents, agent architecture, Software Engineering, Multi-agent systems.

*This material is based upon work supported by the National Science Foundation under Grant No. IIS-9812764. Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Introduction

In a traditional OS (Operating System), the life cycle of a particular process is governed by two basic principles, processes (jobs) have a “lifetime” and files have a “place to live” (Bach 1986). In order to manage the lifetime of a process, a process will pass through several states (e.g. *ready*, *running*, and *sleeping*). In turn the operating system will select, based on some local criteria, which of the ready process can be given some allocation of processor time. Typically, the selection criteria are very simple, since very little is known about a job before it starts.

An agent architecture can be viewed as an operating system for managing the activities of the agent life cycle. However, there are some essential differences between an agent architecture and a general purpose operating system which are listed in Table 1. Figure 1 is a graphical depiction of the Agent action life cycle.

Agent Task Templates do not have a exact equivalent in the traditional OS. Instead the templates represent the possible jobs that may be instantiated by an agent as plans or plan fragments. This representation contains not only a list of tasks that can be requested, but also a description of the relationships between task actions and a description of the execution profile of each task action.

When a request to execute a task is received, the template list is examined to make sure the request is one of the jobs that has been programmed into the framework. If so, then an instantiation of the task is placed on a list of *Actions to be executed* or on a list of *Actions waiting to be enabled*. The first list corresponds to a ready queue while the second is similar to a sleep queue in a traditional OS. An action in execution may do one of two things:

- *complete*, in which case the thread and all other data structures associated with the task are destroyed
- *Sleep awaiting a response* in which case the thread is moved to a queue of other actions waiting and then re-enabled when an appropriate message is received.

The extra level of sophistication that a useful agent framework contains is the ability to reason about which jobs to run next and in what order to run them. The

<i>Traditional Operating System</i>	<i>Agent Architecture</i>
Must handle jobs of any type and input.	All jobs are agent Task requests and input is always via a KQML message.
All actions associated with a job must be completed.	Not all actions must be completed.
Very little is known of a job execution profile before execution.	Execution profile is well characterized.
Mutli-processor actions or network communications must be explicitly programmed.	Agents activities are assumed to be network oriented and support for multi-processors and network communications is built in.
Single or some finite number of processing resources (CPU).	Operates in the Java Virtual Machine (JVM) and has a unlimited number of virtual CPU's for parallel execution.
Number and type of jobs not known at start-up.	Complete list of possible actions initialized with a plan file at start time.

Table 1: Comparing Traditional Operating Systems and Agent Architectures

reasoning ability come from the characterization of the action profile that is programmed into the agent framework. The ability to complete a *set* of jobs before a deadline or to rearrange a task solution in the event of failure is what such frameworks will be able to do that distinguish them from traditional OS's. The following section describes in more detail what an execution characterization includes and how it should be used. It should also be noted in our implementation, using the Java Virtual Machine (JVM) as the underlying (virtual) operating system, DECAF is able to execute many actions in parallel without the explicit instructions of the programmer. While the use of Java and the JVM provide ease of parallel activities and use of multiple processors (if available), the JVM represents essentially a monolithic non-interruptible kernel which implies no hard real-time deadlines can be guaranteed. As more experiments are conducted we hope to place some value on the response time of the JVM and more precisely define the bounds of soft real-time in this environment.

The ability to rapidly prototype agent systems using DECAF has been tested by development of approximately 15 demonstration agent systems in the past two years. DECAF attempts to capture the range of features, processes and interrelationships that occur in computationally intensive AI environments. In DECAF, each capability is represented as a complete task reduction tree (HTN (Erol, Hendler, & Nau 1994)), similar to that in RETSINA (Williamson, Decker, & Sycara 1996), with annotations drawn from the TÆMS task structure description language (Decker & Lesser 1993; Wagner, Garvey, & Lesser 1997). The leaves of the tree represent basic agent actions (HTN primitive actions).

The main thrust of current research is the soft real-time component of the framework. This work is leveraging the work already started using Design-to-Criteria (DTC) scheduling from the University of Massachusetts (Wagner, Garvey, & Lesser 1998). The remainder of

this paper is organized as follows: an introduction to real-time agent scheduling; a description of the DECAF architecture; a description of the characterization of agent execution that DECAF uses to determine schedules and lastly some actual results and development using DECAF.

DECAF Architecture Implementation

DECAF (Graham & Decker 2000) (Distributed, Environment Centered Agent Framework) is an agent architecture for Multi Agent Systems (MAS). Associated with the DECAF architecture is a toolkit which allows a user to specify and program their agent. DECAF builds an operating environment that provides an interface, internal agent scheduling and monitoring in a fashion similar to operating system primitives.

DECAF is the framework used to demonstrate and test the effectiveness of the agent scheduling algorithms. DECAF provides the necessary architectural services of a large-grained intelligent agent (Decker & Sycara 1997; Sycara *et al.* 1996): communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis (Horling *et al.* 1999).

Figure 2 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 2, there are five internal execution modules (square boxes) in the current DECAF implementation, and seven associated data structure queues (rounded boxes).

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

Agent Action Lifecycle

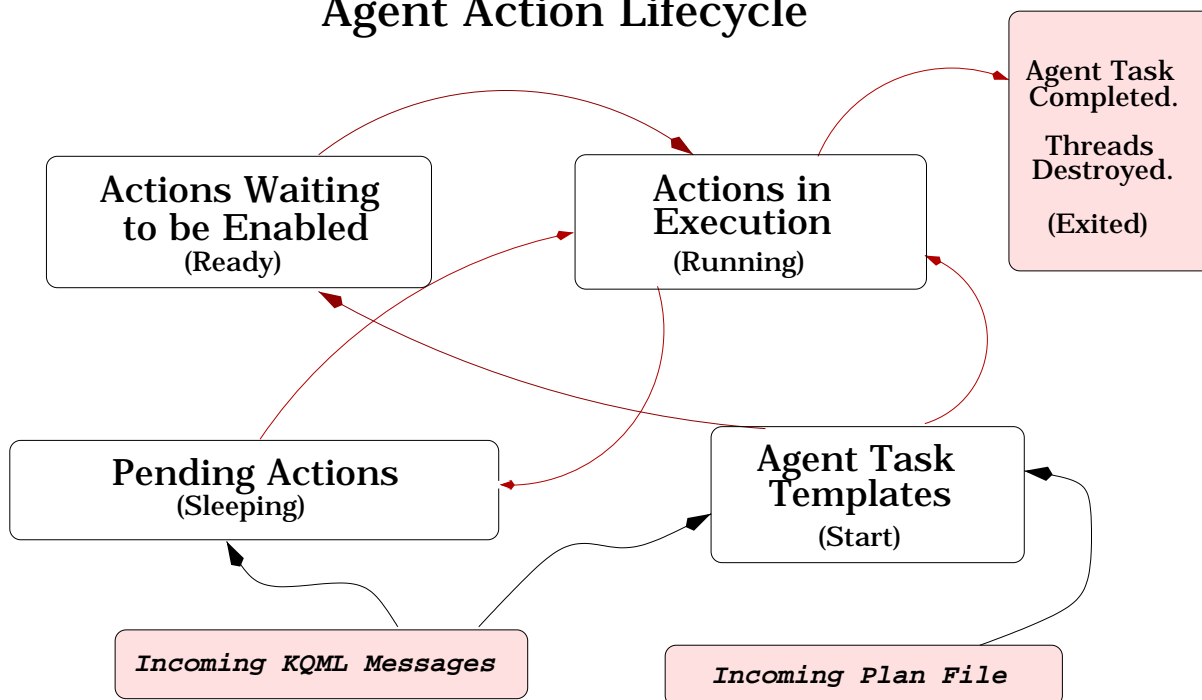


Figure 1: DECAF Agent Action Life cycle

will be added to the *Task Templates Hash table* (plan library) along with the tree structure that is used to specify actions that accomplish that goal.

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 (Rao & Georgeff 1995)) 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 send an error message the Dispatcher is responsible for formatting and send the message.

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.

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.

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: First, the action can

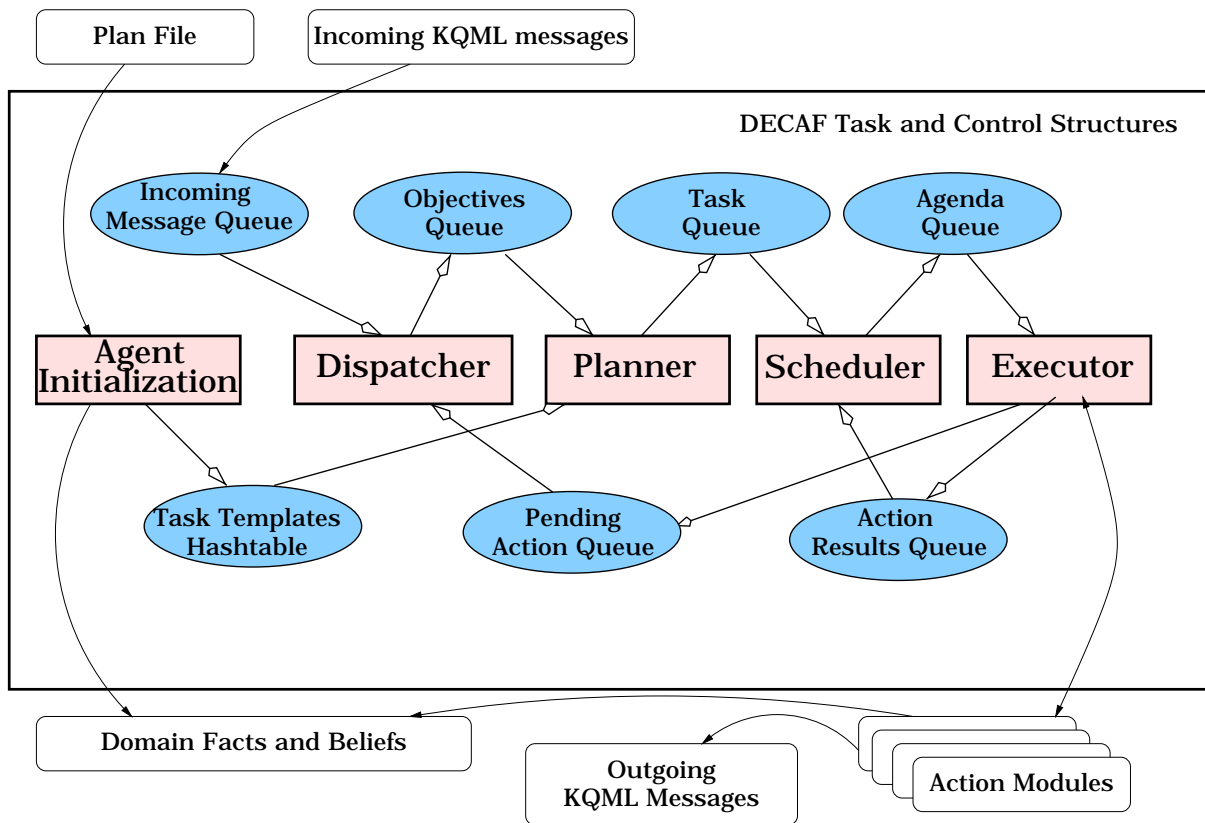


Figure 2: DECAF Architecture Overview

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 second possibility is when the action blocks waiting for agent communication to complete. This occurs when an action sends a message to another agent which requires a response. 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.

More on Scheduling in DECAF

Previous agent architectures have made inroads to real-time AI performance. Guardian (Hayes-Roth *et al.*

1992) is a medical monitoring application built using a blackboard architecture. Guardian uses several features needed to achieve the desired overall system performance (satisficing control, anytime algorithms and dynamic filtering) by having an off-line application maintain the knowledge base. The Phoenix project (Howe, Hart, & Cohen 1990) looks at the design of autonomous systems. The real-time contribution here is the use of a reactive component in an otherwise deliberative agent system. ARTIS (Garcia-Fornes *et al.* 1997) uses a task model based on activities and events. Users (engineers) examine the tasks and apply knowledge they already have of the problem and a prototype is built. The prototype is analyzed off-line for feasibility.

Development of a scheduling algorithm in DECAF will distinguish itself by working in a fashion more like traditional operating systems. The Design-to-criteria (DTC)(Wagner, Garvey, & Lesser 1998) will be leveraged to build the DECAF model. In DTC the idea is to analyze possible schedules and statically evaluate the outcome of each schedule in terms of a user defined utility function. The result is a measure of the “goodness” of a schedule. The best schedule is chosen and then executed. The approach in DECAF will distinguish itself from the basic DTC method in several ways:

- Analysis of possible scheduling paths will be done in dynamically and not off line.

- The emphasis will be on speed. In the event of the failure of a particular action in an execution path, the goal will be to make repairs to the schedule rather than recalculate from scratch
- The set of criteria that was used in (Wagner, Garvey, & Lesser 1998) will be reduced to assist in the speedup
- As actions proceed and are completed, a method of *check-pointing* the path will be employed to help determine exactly how much rework needs to be redone.
- Executing several paths of execution in parallel to reduce possibility of overall failure

DECAF is designed in a modular fashion so that different scheduling algorithms can be tested while keeping the rest of the environment constant. The interface to the scheduling module is well defined and the testing will develop a set of benchmark agent activities to compare. This “drop in” method for analysis has been used to compare three algorithms. (Results are discussed later.)

Development of a Real-time Agent Scheduling Algorithm

In a traditional operating system, jobs are (almost) always run to completion with no prerequisites as to how long they may take. Also, no prior knowledge regarding interprocess communication or resource sharing is present nor is such information required to complete jobs. The result is a simplistic scheduling algorithm which in the case of UNIX, is a set of multi-level feedback queues with first-come-first-served within each queue or in the case of Windows-NT is simply first-come-first-served. Solaris allows for “real-time” scheduling but only under the strictest constraints and definitions. *Real-time* in Solaris is defined as guaranteed access to the CPU in a bounded period of time, but only if no other real-time request has been made and there is no bound on how long the real-time job may take. Real-time actions must be short well defined jobs with no external requests for resources or else the limited definition cannot be met. The Spring operating system (Stankovic & Ramamritham 1989) provides a real-time guarantee by setting aside a time slice for execution of real-time programs.

Of course, such predictable behavior of an agent action is the exception rather than the rule. Searching, in particular, may give better results if more time is allowed. Additionally, search results can vary based on the inputs. When the results of several actions are chained together to achieve a final result, several iterations of the chain may be required to achieve some measure of quality to the ultimate result. This number of iterations may be limited by time which in turn puts a limitation on the time-slice allocated to each piece of the chain.

Because of the difficulties in predicting the expected performance of algorithms for solving AI problems, the

worst case performance for these algorithms and the cost of scheduling required task structures, some limitations in complexity are required to achieve real-time results in solving these problems. In the context of agent execution, *real-time* is defined to mean that an agent will statistically (i.e. on average) achieve the required quality of result in the required time but no guarantee is made about any particular task (Garvey & Lesser 1994). This notion of real-time is sometimes referred to as “soft” real-time as compared to the “hard” real-time context of the Solaris OS.

In an agent architecture, developing solutions to tasks is an imprecise act which is one of the particular features that make agent solutions attractive. The fact that a solution is not hard coded and that the failure of a selected path does not result in the failure of the whole solution makes the agent solution more robust and at the same time makes the scheduling and execution of the agent actions difficult. A given solution involves many separate actions and multiple different execution paths for achieving a particular task. The resulting problem of scheduling the various tasks (and potentially rescheduling in the face of a failure) is intractable.

The DECAF scheduler will apply heuristics to the problem to obtain scheduling solutions. The resulting schedule will not be optimal (in general) since obtaining an optimal solution is an NP-complete problem. However, the intent is to develop a solution that satisfies the user criteria with regard to a limited set of characteristics applied to each agent action. The direction that is being taken is to embody “soft” real-time into the component architecture of DECAF by allowing an agent to program and get *commitments* (Jennings 1993) from the architecture.

User Defined Scheduling Criteria

Programming an agent, as stated earlier, involves developing a Hierarchical Task Network of tasks and actions. Associated with each task or action is a list of user definable criteria that will be used to define a schedule.

- Behavior Profile (Agent Actions only)
- Task and Action relationships and deadlines
- Characteristic Accumulation Function (Tasks only)
- Utility Function (Tasks Only)

The following sections discuss these items and their use and provide the formal model for agent performance in DECAF.

Behavior Profile Each agent action will have a behavior profile attached to it. A behavior profile can be viewed as a vector of characteristics $C = \{ C_1, C_2, \dots, C_n \}$. These characteristics relate to the performance of the action itself and not to the result produced by the action. For example, the actual outcome of an action may be *fail* but the quality with regards to the performance of the action in achieving that result may be very high in terms of what the user expected. From

a mathematical perspective, the set of characteristics to be considered may be any size. However, one of the heuristics applied for our solution is to limit the vector to three particular items, cost, quality, and duration. In this context cost and duration are exactly what you would expect (time and money!) and as such accumulation functions are easily calculated. Quality is a more nebulous item which has meaning only in the users context. A quality of 10 for an action that finds a low price airline ticket may be extremely low while the same value for an agent that responds to changes in the price of IBM stock may be very good. The quality of performance only has meaning when considered in the context of the pool of actions to be scheduled. In determining optimal performance, the focus of the research is on the values of the various characteristics of the agent actions (cost, quality and duration in our case) and not the value of the *result* of the task, i.e. we distinguish between *domain-level* reasoning (the and *control-level* (finding an optimal execution of agent actions) reasoning. It is in this way that we can reason about good performance versus bad.

Relationships and Deadlines While the performance of an action may not affect subsequent actions, in terms of the scheduling problem the dependencies or relationships between actions is very important. If action A_i must complete before action A_j can begin, then the time to complete the sequence of actions is clearly the sum of the time to complete both. If A_i and A_j are not related and can be run in parallel then the minimum time to run is the time of the longest action. In DECAF, such relationships are explicitly programmed by the developer using the DECAF *Plan Editor* This approach differs from TÆMS in that there is only one type of relationship defined; the idea of “happens before”. In TÆMS there is the notion of “soft” relationships, such that while it is not essential that action A_i run before action A_j , there is potentially a better quality result if this does occur. Only “hard” relationships are reasoned about in DECAF in order to reduce computational complexity.

Characteristic Accumulation Function Each task will have associated with it a Characteristic Accumulation Function (\mathcal{A}). The purpose of the \mathcal{A} is to inform the scheduler how the performance metrics of the subtasks (or actions) are to be gathered. The possible functions that the user may specify are AND,OR,XOR, and SUM. If the \mathcal{A} is AND, that means that all of the subtasks must be performed before making decisions. On the other hand a \mathcal{A} of OR would indicate that any of the subtasks are sufficient to make a determination. The indication here, assuming sufficient external resources, would be to run all of the subtasks in parallel and the first one finished will be used while other results will be ignored.

This approach will leverage scheduling work already done by (Wagner, Garvey, & Lesser 1998). In fact one of the initial benchmarks for comparison will be to com-

pare schedules to those developed by the previous work. The premise of (Wagner, Garvey, & Lesser 1998) is that the goodness of a solution is dependent on the client complex objectives and that different client’s have different objectives. The characterization of \mathcal{U} , \mathcal{A} , and \mathcal{C} are quantifications of those objectives. Also, the scheduler must react dynamically to events. The exact calculus and semantics of the operators (AND,OR,XOR, and SUM) are still under consideration. There may be a proposed schedule but in the event of a failure of a node the schedule must be recomputed. Thus scheduling activities will be interleaved with planning and execution of modules.

Utility Functions It is not enough to simply have an agent accomplish a goal. For example, if the goal is to get a taxi ride from point A to point B, there are an almost infinite number of ways to accomplish the goal. Presumably however, the user has a desire to make the ride as short as possible, or as cheap as possible or even as relaxing as possible. These are descriptions of **utility** for the agent (Russell & Norvig 1995).

A task utility function (\mathcal{U}) is an assignment by the user of some idea of “goodness” of the actions that make up the task. While the values for the performance characteristics of an an action may be fixed how the user chooses to use that measure may vary. For example, the cost of looking up a ticket price at web site W_1 is always \$1. To look up the price at another web site W_2 is always \$5. However, the actual price of the ticket found at W_1 is always greater so the quality associated with the result is lower.

An example of a utility function of the task that can retrieve a ticket price from either site may be “The Minimum cost of finding the ticket price”. In our example, that means the result produced from W_1 will be used and the associated action scheduled. A more complex utility function may say that the cost is not relevant but quality is the most important.

An agent that possesses such an *explicit* utility function can make rational decisions about the utilities of different courses of actions. This will be one criteria for selecting on schedule for a sets of actions over another.

Current Activities and Research

The DECAF architecture is designed so particular aspects of agent development can be singled out and examined. The one project featured here is the work relating to real-time agent scheduling.

Real-Time Schedule Design Criteria

The major research project using DECAF underway is development and evaluation of a real-time scheduling component. This scheduling component features:

- Use of agent deadlines and commitments to achieve the soft real-time performance defined earlier.
- A faster heuristic for achieving the “best” result from a set of agent sequences.

- Use of fast on-line plan evaluation for determining best results.

Taking this approach to the real-time issues represents an improvement in recent work for two reasons:

- The dynamic in-line evaluation of plans combines a deliberative and reactive component to scheduling activities not previously used.
- Extensive use of parallel execution will allow the reactive component to respond to failure in a strictly more timely fashion.

To evaluate the success of the new scheduling algorithm there will be three scheduling modules inserted in the DECAF framework. First is a simple first-come-first-served (FCFS) routine, second will be a DTC algorithm modified to run in-line and the third will be the new algorithm of this development. The new algorithm has a preliminary acronym of DRU (Dynamic Real-time Update). DRU will evaluate a plan and will select several possible solutions and begin execution of all of them. How many solutions to select is still under analysis but a reasonable first estimate is that 10% of possible plans will be used. The idea is that this selection will run very quickly compared to the off-line DTC analysis. This will also give a faster recovery in the event of a failure since an alternate plan is already in execution and no further analysis is needed.

Currently, the FCFS has been implemented and an off-line version of DTC has been implemented with expected results. The DTC schedules produced have been of higher quality than FCFS and somewhat faster. Also implemented is a resource scheduling algorithm which uses an algorithm very similar to the agent action scheduling. It is anticipated that DRU will produce similar quality but in less time and be able to recover from agent node failures in a more robust fashion. Final results are not complete yet but preliminary results are based on two criteria; the quality of the schedule and the time to determine the schedule. We start from the premise that DTC will always select the best schedule and use that as a baseline. Currently there is no data to report about the quality of schedules picked by DRU. However, the time analysis looks promising. For example, to evaluate a plan with 15 schedules takes DTC 2-4 seconds while DRU can determine a set of schedules to execute looking the same plan in less than 500 milliseconds.

DECAF Architecture Performance

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 get its own thread and 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 several other methods, including: binary semaphores, counting semaphores, condition variables and reader/writer locks. The basis of this language enhancement comes from (Hartley 1998).

There is a (as yet unpublished) technical report which details the overall performance of the DECAF architecture as is summarized in part here:

- **Threaded execution.** Each module of DECAF and each agent action is executed as a separate thread in the JVM. On a single processor machine, a threaded version of DECAF ran 300% faster than a non-threaded version.
- **Effects of Multiple processors.** The actual performance benefit using many CPU's will depend greatly on the actual plan and the computing involved. In a compute intensive environment we would expect the greatest benefit (as compared to an I/O intensive application). Our results showed this to be the case with improvements from 15% to 100% when when tested on 1, 12, 20, and 40 CPU's.
- **Schedule Evaluation.** The premise of DRU is that we will be able to execute several plans in the time it takes DTC to evaluate and run one plan due to efficiencies in the architecture. This has shown to be accurate. The time to run ten schedules was within 10% of the time to evaluate and run the one best schedule determined by DECAF.

Current MAS Development and Architecture Evaluation

DECAF is being presented as a general purpose agent architecture much as UNIX is presented as a general purpose operating system. To that end, DECAF has been used in classes on MAS for development of agent systems. The validity of the use of DECAF as an agent development platform has been proven over the two years. Some completed projects developed using DECAF include:

- **Information Extraction Agent.** This is a front end for a generic database access. It is easily adapted to a specific database for extraction of information.
- **Virtual Food Court.** This MAS is a modeling of the interactions that are required for restaurants, diner, and suppliers to operate.

- **Resource Management.** Each agent is capable of requesting and using resource in the agent community. This work features a set of routines for resource management and development of a deadlock detection agent.
- **Middle-ware.** Agent that provide services to the agent community are known as middle agents (Decker, Sycara, & Williamson 1997). A *Match-Maker* and a *Broker* have been developed for DECAF

Conclusions and Summary

We have shown the fundamental outlines of an agent architecture based on operating system principles that incorporates the concept of soft real-time. This architecture is being used to develop working MAS models and as a research platform for the development of an agent scheduling algorithm. In the next several months, the details of the algorithm will be put in place and precise measurements of the effectiveness of the work will be shown.

References

- Bach, M. 1986. *Design of the UNIX Operating System*. Prentice-Hall.
- Decker, K. S., and Lesser, V. R. 1993. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 217–224.
- Decker, K. S., and Sycara, K. 1997. Intelligent adaptive information agents. *Journal of Intelligent Information Systems* 9(3):239–260.
- Decker, K. S.; Sycara, K.; and Williamson, M. 1997. Middle-agents for the internet. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 578–583.
- Erol, K.; Hendler, J.; and Nau, D. 1994. Semantics for hierarchical task network planning. CS Technical Report TR-3239, University of Maryland.
- Garcia-Fornes, A.; Terrasa, A.; Botti, V.; and Crespo, A. 1997. Analyzing the schedulability of hard real-time artificial intelligence systems. *Engineering Applications in Artificial Intelligence* 10(4).
- Garvey, A., and Lesser, V. 1994. A survey of research in deliberative real-time artificial intelligence. *The Journal of Real-Time Systems* 6.
- Graham, J. R., and Decker, K. S. 2000. Towards a distributed, environment-centered agent framework. In Jennings, N., and Lespérance, Y., eds., *Intelligent Agents VI — Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin. In this volume.
- Hartley, S. J. 1998. *Concurrent Programming, The Java Programming Language*. Drexel University: Oxford University Press.
- Hayes-Roth, B.; Washington, R.; Ash, D.; Collinot, A.; Vina, A.; and Seiver, A. 1992. Guardian: A prototype intensive-care monitoring agent. *Artificial Intelligence in Medicine* 4:165–185.
- Horling, B.; Lesser, V.; Vincent, R.; Bazzan, A.; and Xuan, P. 1999. Diagnosis as an integral part of multi-agent adaptability. Tech Report CS-TR-99-03, UMass.
- Howe, A. E.; Hart, D. M.; and Cohen, P. R. 1990. Addressing real-time constraints in the design of autonomous agents. *The Journal of Real-Time Systems* 2(1/2):81–97.
- Jennings, N. R. 1993. Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review* 8(3):223–250.
- Rao, A., and Georgeff, M. 1995. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems*, 312–319. San Francisco: AAAI Press.
- Russell, S., and Norvig, P. 1995. *Artificial Intelligence, A Modern Approach*. Saddle River, NJ: Prentice Hall.
- Stankovic, J. A., and Ramamritham, K. 1989. On using the spring kernel to support real-time ai applications. CS technical report tr89-25, Univ. of Massachusetts.
- Sycara, K.; Decker, K. S.; Pannu, A.; Williamson, M.; and Zeng, D. 1996. Distributed intelligent agents. *IEEE Expert* 11(6):36–46.
- Wagner, T.; Garvey, A.; and Lesser, V. 1997. Complex goal criteria and its application in design-to-criteria scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*.
- Wagner, T.; Garvey, A.; and Lesser, V. 1998. Criteria-Directed Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling* 19(1-2):91–118. A version also available as UMASS CS TR-97-59.
- Williamson, M.; Decker, K. S.; and Sycara, K. 1996. Executing decision-theoretic plans in multi-agent environments. In *AAAI Fall Symposium on Plan Execution*. AAAI Report FS-96-01.