DECAF - A Flexible Multi Agent System Architecture *

John R. Graham
Coastal Carolina University
Conway, SC, 29582, USA

Keith S. Decker, Michael Mersic
University of Delaware
Newark, DE, 19716, USA

September 1, 2000

Abstract.

The first wave of agent implementation toolkits focussed mostly on providing APIs for agent communication. We believe that new toolkits should focus on the public dissemination of complete agent architectures that provide significant value over building software agents from scratch. 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. DECAF is based on the premise that execution of the actions required to accomplish a task specified by an agent program is similar to a traditional operating system executing a sequence of user requests. In the same fashion that an operating system provides an environment for the execution of a user request, an agent framework provides the needed environment for the execution of agent actions. The agent environment includes the ability to communicate with other agents, efficiently maintain the current state of an executing agent, and select an execution path from a set of possible execution paths so as to support persistent, flexible, and robust actions.

From a research community perspective, DECAF provides a modular platform for evaluating and disseminating results in agent architectures, including communication, planning, action scheduling, execution monitoring, coordination, and learning. By modularizing the design of the software, researchers can attack and analyze specific issues in agent development, coordination and planning without disturbing other parts of the architecture.

From a user/programmer perspective, DECAF distinguishes itself by removing the focus from the underlying components of agent building such as socket creation, agent communication, and efficient implementation of complex architectural details. 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, and various supporting middle-agents.

This article discusses the high level architecture of DECAF long with comparisons to operating systems architecture and other agent frameworks; descriptions of supporting middle agents and development tools; and an analysis of projects already developed using DECAF and some performance benchmarks from DECAF.

* This material is based upon work supported by the National Science Foundation under Grant No. IIS-9812764.

1. Introduction

DECAF (Distributed, Environment-Centered Agent Framework) is an agent toolkit which allows a well-defined software engineering approach to building multi-agent systems. The toolkit provides a platform to design, develop, and execute agents to achieve solutions in complex software systems. DECAF provides the necessary architectural services of a large-grained intelligent agent [11, 42]: communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis [28]. This is essentially, the internal "operating system" of a software agent, to which application programmers have strictly limited access.

The earliest publically available agent implementation toolkits focussed mostly on providing increasingly well-thought-out APIs for agent communications [29, 35]. In order to build agents, programmers needed to piece together those APIs to create some kind of complete agent architecture from scratch. While this made supporting different research goals easy, it also made it harder for students, multi-agent application programmers, or researchers interested in only some agent architectural components to develop thier ideas quickly and efficiently. The DECAF project attempts to meet the needs of these kinds of users while remaining a viable research platform for advanced planning, action scheduling, and multi-agent coordination techniques.

For non-researchers, the focus of an agent toolkit should be on programming agents and building multi-agent systems, and not on designing new internal agent architectures from scratch for each project.[1] 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 (potentially automated via a 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 TÆMS task structure frameworks [46, 8]. In particular, DECAF uses RETSINA-style automated data flow for directing plan execution.

Another important goal for an agent toolkit is to support that which makes agents different from arbitrary software objects: flexible (reactive, proactive, and social) behavior [47]. DECAF encourages the programming of tasks for robust and persistent goal achievement [22], supporting programers in detecting problematic outcomes, and allowing multiple ways to carry out tasks to be chosen dynamically at runtime. For example, each action can also have attached to it a performance profile (description of action duration, cost, and

[1] Designing new agent architectures is certainly an important *research* goal, but fraught with peril for beginning students and programmers wanting to work with the agent concept.

quality) 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 TÆMS and the design-to-time/design-to-criteria scheduling work at UMASS [20, 43].

The other goals of the architecture are to develop a modular platform suitable for our research activities, allow for rapid development of third-party domain agents, provide a means to quickly develop complete multi-agent solutions using combinations of domain-specific agents and standard middle-agents [12], and to take advantage of features of the JAVA programming language to provide an efficient development framework.

From a multi-agent systems (MAS) viewpoint, researchers attempting to develop new agents are faced with the problem of constructing a robust environment for executing their agent tasks. This environment must be able to use and understand network and communication protocols, adapt in the face of failure, and provide a platform for development of the agent tasks themselves. Then the tasks must be organized (programmed) to provide the "intelligence" of the agent code, and multi-agent activity must be supported and coordinated via scheduling and communication protocols. As a toolkit, DECAF helps in prototyping MAS by taking care of certain common details (via reusable behaviors [10]) and providing standardized, domain-independent or easily customizable middle agents [12]. Reusable behaviors include things such as Agent Name Server registration and deregistration, Agent Management protocols, and service negotiation via middle agents such as matchmakers. DECAF middle agents include agent name servers, matchmakers, brokers, information extraction agents, web proxies, and agent management agents.

2. Features of DECAF

DECAF provides an end-to-end environment for the execution of agent tasks. In this sense DECAF has been developed with similar functions of an operating system where jobs are entered and executed to completion without user intervention. Also similar to an operating system, a set of system services for ease of software development have been provided.

In order to support the development of agents, other tools have also been developed to support agent operations and software design. Currently the tools include the following components:

The **The Plan Editor** is a GUI interface that allows graphic programming of agent tasks and actions. The Plan Editor allows editing of features needed to reason about scheduling activities, representation of message sending, use of library plans, and control flow logic.

Middle Agents have been developed to support common multi-agent activities. A middle agent is an agent that facilitates agent operation while not directly related to completing a specific task. The **Matchmaker** serves as a "yellow pages" to assist agents in finding services usual for task completion. The **Broker** agent acts as a "white pages" directory to assist agent with collections of services. A **Proxy** agent allows web page Java applets to communicate with DECAF agents that are not located on the same server as the applet. The **Agent Management Agent (AMA)** allows MAS designers a look at the entire running set of agents spread out across the internet that share a single agent name server. This allows designers to query the status of individual agents and watch or record message passing traffic.

The **Agent Name Server (ANS)** is an essential component for agent communication. It works in a fashion similar to DNS (Domain Name Service) by resolving agent names to host and port addresses.

Specifically, DECAF represents an advancement in the discipline of implemented, publically available run-time agent architectures that focusses on the following areas:

Concurrent processing. DECAF is a highly threaded architecture running separate threads for each action and each internal module. The advantage of this is the ability to scale the architecture of a single agent on a multiprocessor platform with few changes to user code[2]

General Problem Domain. DECAF has been designed without any specific domain knowledge, and should be useful in most traditional multi-agent applications. For example, DECAF is currently being used for applications in bioinformatics research, chemical engineering, wildlife management, and social simulations. We also use DECAF for teaching multi-agent systems concepts and protocols[32]

Usefulness as a research platform. The modular design of DECAF allows independent development of algorithms for various areas of research. Currently, research in planning[26], action scheduling[24] and GPGP (Generalized Partial Global Planning) are underway using the DECAF platform (See Section 8).

[2] However, threaded actions are harder to write than ones guarenteed to execute sequentially.

One problem with comparing agent architectures stems from the problem of an inconsistent or non-standard way of defining an agent. Nonetheless, an agent architecture needs specific capabilities and features to be useful. DECAF was designed to address some specific issues in an agent architecture. *Problem Domain:* Is the architecture designed to handle problems in one specific domain (financial planning, air, traffic control, ...) or can it be programmed to handle problems in any domain? *Software Engineering:* Does the architecture require a complex interface? Does the programmer have to deal with all aspects of agent behavior, (planning, scheduling, communication, execution, ...) or only with the detailed aspects of agent actions? *Internals vs. Interactions:* Do multi-agent system builders focus on the details of the architecture of each agent, or on the interactions between the agents? *Suitability as research platform:* Can each feature of an architecture be evaluated and tested separately or are they integrated together? *Usability:* Is the architecture written in a portable language and distributed with source code for ease of maintenance, flexibility and adaptability to specific tasks. Does the architecture provide a toolkit that makes a total development and execution package available to users? *Threading and Scalability:* Does the architecture scale to run across many separate computers? To make use of multiple processors on one host? Make use of unused cycles during I/O on single CPU hosts? *Reasoning:* Does the architecture provide any reasoning for planning? For Scheduling?

3. The DECAF Approach

DECAF is a type of hybrid agent architecture [47]. The Plan Editor builds a symbolic plan which is then used by the *Planner* to engage in some reasoning about how to best arrange a selection of agent actions to accomplish a goal, including possible run-time alternative execution paths. The *Scheduler* chooses an appropriate execution path, given the current set of plans being attempted. Finally, the *Executor* of DECAF is a reactive component without complex reasoning to carry out the chosen actions.

Functionally, DECAF is based on RETSINA [42, 10, 11, 46, 45] and TÆMS [8, 43]. TÆMS provides a framework useful for agent coordination by defining how agent actions and tasks relate to each other, as well as a calculus for describing various plan alternatives that can be chosen at run-time by a action scheduling component. RETSINA provides the idea of adaptability by allowing multiple outcomes, and reactive data flow constructs.

However, DECAF has been structured to provide a platform for rapid development of agents and as a platform for researching specific areas of agent interaction. DECAF is written in Java and makes extensive use of the Java threads capabilities to improve performance—each agent subcomponent

Table I. Agent Architecture vs. Traditional Operating Systems

Traditional Operating System	*Agent Architectures*
Must handle jobs of any type and input.	All jobs are agent Task requests and input is 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 a virtual machine 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.

runs concurrently within its own thread. DECAF supports a general way to map KQML messages such as ACHIEVE to arbitrary plan fragments. DECAF uses an HTN representation that is a hybrid of TÆMS and RETSINA.

3.1. Agent Architectures and Operating Systems

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" [1]. several states. The operating system will select, based on some local criteria, which of the ready processes 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 acts as an operating system for managing the activities of the agent life cycle. DECAF Manages the actions of an agent in a fashion very similar to traditional operating systems. There are some essential differences between Agent architectures and a general purpose operating system listed in Table I.

The extra level of sophistication that DECAF contains is the ability to reason about which jobs to run next and in what order to run them. The 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 DECAF is able to do that distinguishes it from the traditional OS. The following section describes in more detail what the execution characterization

includes and how it will be used. It should also be noted that by 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 agent programmer.

4. DECAF Theory

Just as there is no clear consensus as to precisely which characteristics best define the notion of an agent, there is no clear consensus about which combinations of information are best suited to characterizing rational agents. 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 [37]—that of balancing reconsideration of activities in a dynamic, real-time environment with taking action [39]. 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 "three tier" models [18, 40, 3] have proven extremely useful (here, Planning, Scheduling, and Execution Monitoring). 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[3].

In keeping with this notion of BDI we present a formal model of DECAF component relationships. This representation defines the semantics of agent performance competencies and reasoning about their use in isolation and in combination. The DECAF architecture implements five basic functions:

- *Initialization* - $\mathcal{I}(\mathbf{PF}) = \{\mathbf{TT}\}$ where the initialization function $\mathcal{I}$ takes a plan file $\mathbf{PF}$ as input and produces a *set* of task templates $\mathbf{TT}$ as output. The set of task templates is a definition of the *capabilities* of this particular agent .
- *Dispatching* - $\mathcal{D}(\mathbf{M}) = \mathbf{O}$ where the dispatcher function $\mathcal{D}$ takes a KQML message $\mathbf{M}$ as input and produces a new objective $\mathbf{O}$ as output. The dispatcher can run in one of two modes. First, if the incoming message represent a new request then a new objective is created. Second, if the incoming message is a response to a previous message as part of an ongoing conversation, the output from the dispatcher will be the objective that was previously created when the original message was sent.

[3] See also the discussions of plans and plan actions as intentions by Cohen and Grosz & Kraus [6, 25].

- *Planning* - $\mathcal{P}(\mathbf{O}, \{\mathbf{TT}\}) = \mathbf{TQ}$ where the planning function $\mathcal{P}$ takes an objective $\mathbf{O}$ and a set of task templates for that objective $\{\mathbf{TT}\}$ as input and produces an instantiation of the appropriate task template, known internally as a task queue object $\mathbf{TQ}$ as output.

- *Scheduling* - $\mathcal{S}(\{\mathbf{TQ}\}) = \{\mathbf{A}\}$ where the scheduling function $\mathcal{S}$ takes a set of task queue objects $\mathbf{TQ}$ as input and produces an agenda $\mathbf{A}$ as output. The agenda is a set of actions to perform. Simultaneously, $\mathcal{S}$ notes when actions have been completed and more actions can be enabled as a result.

- *Execution* - $\mathcal{E}(\{\mathbf{A}\} = \{\mathbf{IN}\}$ where the execution function $\mathcal{E}$ takes a set of enabled actions $\mathcal{A}$ as input and produces a set of "intentions" (low level commitments to very specific courses of action—a Java procedure invocation).

In summary, the set of actions to be executed by DECAF is:

$$\{\mathbf{IN}\} = \mathcal{E}(\mathcal{S}(\mathcal{P}(\mathcal{D}(\mathbf{M}), \mathcal{I}(\{\mathbf{PF}\})))) \tag{1}$$

In colloquial terms, an instantiation of the architecture takes a plan file, $\mathbf{PF}$, and waits for a message, $\mathbf{M}$. When the message arrives, there are three possibilities:

- The message specifies a task that is not in the capabilities of the plan file. In this case the action taken is to send an error message back to the sender of the message.

- The message specifies the start of a new task not previously requested. In this case, the architecture will follow each of the steps above and produce a set of actions to complete the task.

- The message is in response to a message sent to assist in completion of an ongoing task. In this case, the dispatching, planning and scheduling functions are skipped and the content of message is reported to the awaiting task.

It is important to note that the theoretical definition does not indicate a key implementation feature of the architecture which is concurrency. Each component is implemented in the Java code with a separate *concurrently* executing thread. This means that to process one message does not block processing of more messages as they arrive. If the architecture is running on a multi-processor machine many messages may be in process at the same time. On a single processor machine, the order or execution will be the order in which the messages arrive but proccessing will make use of unused cycles

by using the threading mechanism. The scheduler does not start fresh each time, but rather attempts to integrate new actions into the existing agenda.

5. The DECAF Toolkit

The basic operation of DECAF requires three components: an agent name server (ANS), an agent program (plan file), and the DECAF framework itself.

The purpose of the ANS is similar to most name servers such as DNS (Domain Name Server). The idea is like looking up someone's name in the phone book (white pages) and then making a call. 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; currently DECAF uses the RETSINA ANS protocol.

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 or goals) and a collection of actions that may be planned and executed to achieve the objectives. These capabilities can correspond to classical AI black-and-white goals or "worth-oriented" objective functions over states [38, 45]. Currently, each capability is represented as a complete task reduction tree (HTN [16]), similar to that in RETSINA [45], with annotations drawn from the TÆMS task structure description language [8, 43]. 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 Java statements but are a sequence of actions connected in a manner that will achieve the goal[4]. Actions are reusable in any sequence for the achievement of many goals. Plan annotations can include alternative subgoals ("OR": choose at least one, extras don't help; "XOR": choose only one subgoal; "SUM": choose at least one, but it's best to do them all if possible) [24].

The Plan-Editor interface for creating plan files was influenced by work such as the software component editor for ABE [27] and the TÆMS task structure editor for the Boeing MADEsmart/RaDEO project [2]. In the Plan-Editor, a capability is developed using a HTN 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 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

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

or more *outcomes* [46]. The provisions to a node may either be forwarded from the provisions of a parent task, or come from the outcomes of different actions. No action will start until all of its provisions have been supplied by an outcome being forwarded from another node (this may be an external node representing some non-local task and the reception of a KQML or FIPA 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.

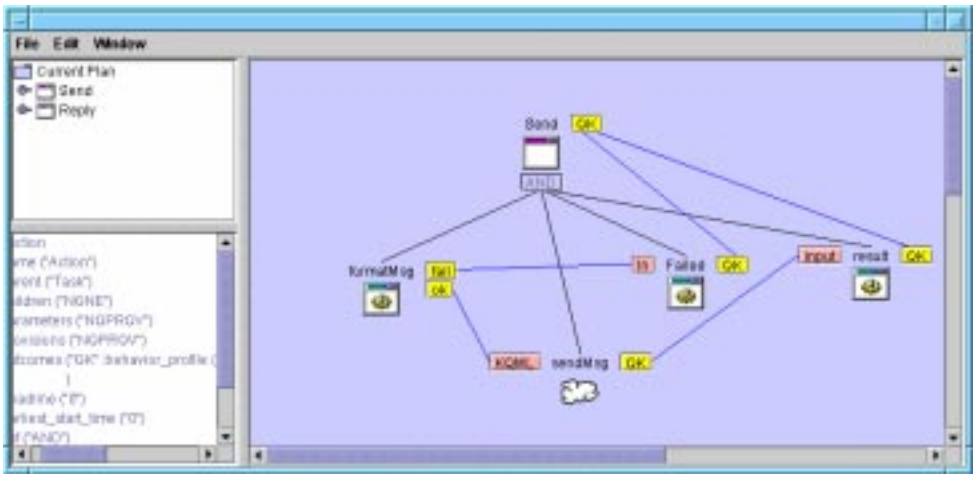

Figure 1. Sample Plan File

A node may have multiple outcomes and is considered complete as soon as one outcome has been provided. The *outcomes* represent a complete *classification* or partition of the possible results. By connecting different node outcomes to different downstream input provisions in the Plan-Editor, conventional looping or conditional selection can be created. A snapshot of a sample Plan-Editor session is shown in Figure 1.

In Figure 1, *Send* is the name of the root task, *formatMsg*, *Failed* and *result* are the actions. The cloud *sendMsg* represents a non-local task accessed via inter-agent communication. Task input provisions are on the left side of icons, and outcomes on the right side. When the task Send is invoked, formatMsg will be enabled to execute and depending on the outcome (*ok* or *fail*) either [sendMsg followed by result]; or [Failed] will execute. When either result or Failed is complete, the outcome (*OK*) is provided to the root task indicating that the goal has been completed. Williamson provides formal details of provision/outcome based HTN planning in [46].

6. DECAF Implementation

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

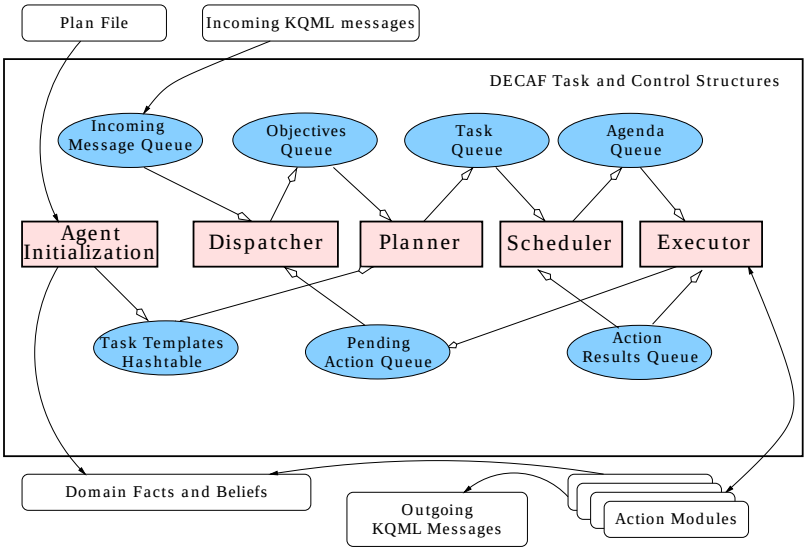

Figure 2. DECAF Architecture Overview

6.1. 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(s) as described above. Each task reduction specified in the plan file will be added to the *Task Templates Hash table* (plan library).

Next the plan may make use of a *Startup* module. The Startup task of an agent might, for example, build any domain database/knowledgebase needed for future execution of the agent, or register with a Matchmaker. Any 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. 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. Finally, the Agent Initialization Module does is register with the ANS and set up all socket and network communication.

6.2. DISPATCHER

Agent initialization is done once and then control is passed to the *Dispatcher* which waits for incoming KQML messages which are be placed on the *Incoming Message Queue*.

An incoming message contains a KQML *performative* and its associated information become an objective indicatng which capability within the agent is to be accomplished. An incoming message can result in one of three actions by the dispatcher.

- 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.

- The message indicates 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[37]) and placed on the *Objectives Queue* for the Planner. The dispatcher assign a unique identifier to this message which is used to distinguish all messages that are part of the new conversation.

- 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 is responsible for formatting and send the message.

6.3. PLANNER

The Objectives 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. The initial, top-level objectives are roughly equivalent to the BDI "desires" concept[37], while the expansion into plans is only part of the traditional notion of BDI "intentions", which for DECAF is divided into three reasoning levels, planning, scheduling, and execution. The *Plan scheduler* sleeps until the Objectives Queue is non-empty and will go back to sleep when the queue is empty. The purpose of the Plan 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 objectives queue is checked any time a provision becomes available to see which actions can be executed now.

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 queue 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.

6.4. SCHEDULER

The *Scheduler* waits until the Task Queue is non-empty. The scheduling functions are actually divided into two separate modules; the *Task Scheduler* and the *Agenda Manager*.

The purpose of the Task Scheduler is to evaluate the HTN task structure to determine a set of actions which will "best" suit the users goals. The input is a task HTN will all possible actions, and the output is a task HTN pruned to reflect the desired set of actions.

Once the set of actions have been determined, the Agenda Manager (AM) is responsible for setting the actions into execution. This determination is 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.

The other responsibility of the AM is to reschedule actions when a new task is requested. Every task has a window of time that is used for execution. If subsequent tasks can be completed while currently scheduled are running then a commitment is made to running the task on time. Otherwise the AM will respond with an error message to the requester that the task cannot be completed in the desired time frame.

6.5. 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. 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 distributes the result as provisions 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.

7. Evaluation: Architecture

The DECAF Agent architecture is a general purpose agent development platform designed specifically to support concurrency, distributed operations, high level programming paradigms and high throughput. The architecture is highly threaded to adapt itself to multi-processor architectures. This section will show four specific tests of DECAF performance: scale across multiple processors; threaded modularity; use of idle cycles in the JVM architecture; throughput and network behavior.

7.1. SCALABILITY

Publications dealing with scalability of MAS's (multi agent systems) are relatively few. In the context of multi agent communities, MAS's will need to scale in a number different dimensions [36]: when the number of agents involved increases on a single platform; when the number of agents involved increase across multiple platforms; when the size of the data the agents are operating with increases; when the diversity of agents in the community increases. When the first two items are increased the the resulting performance can be determined in more or less standard metrics; memory usage, scheduling and swapping overheads, and relative speed.

Testing scalability is a matter of observing results when the underlying architecture (such as number of CPU's) is varied or the software architecture (threaded vs. non-threaded) is changed.

7.1.1. *Testing the Threaded Architecture*

By default, DECAF will parallelize and thread as much execution as possible. This requires a large amount of synchronization between methods. The time for the synchronization should be computationally insignificant when compared to a non-threaded version of the execution module. This test gives a measure of the benefit of a threaded architecture. The benefits of threading vary greatly depending on the type action being performed. I/O bound activities show much greater benefit (even on single processor machines) than compute bound actions.

In order to scale a complex task it is essential make sure the Java Virtual Machine (JVM) makes use of threads and multiple processors in the obvious fashion. To test this, we wrote a computationally complex agent action. If the action is run many times in the accomplishment of the task, we would expect to see direct benefit from threading and multiple processors.

Figure 3 shows that in the absence of threading the number of processors is of little benefit. As the number of processors is increased, there seems to be little time improvement.

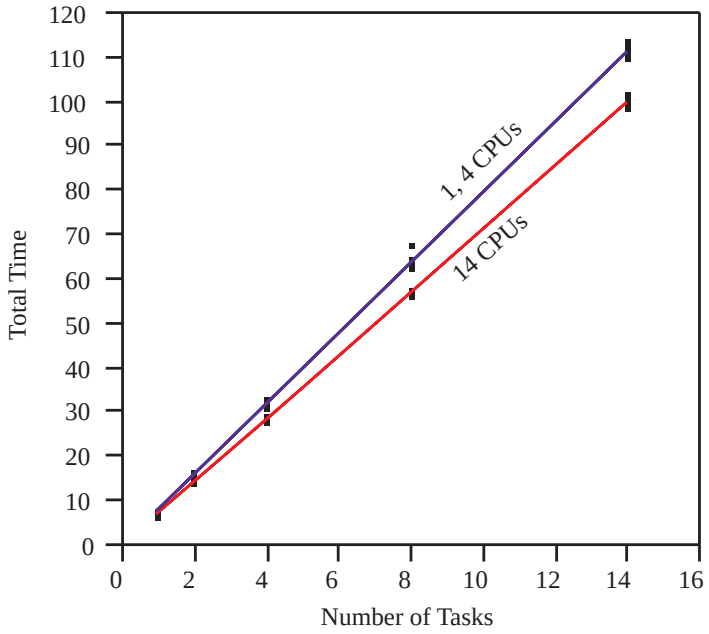

Figure 3. Non-Threaded Execution Results

The test to verify threaded execution behavior in the JVM is the same test as above but with a threaded version of DECAF. In this case each action spawn a new thread. Figure 4 shows that if there is only one processor, there is nothing to be gained by multi-threading. The difference between the values shown in Figure 3 and Figure 4 is the result of threading and shows that the JVM works in the expected manner.

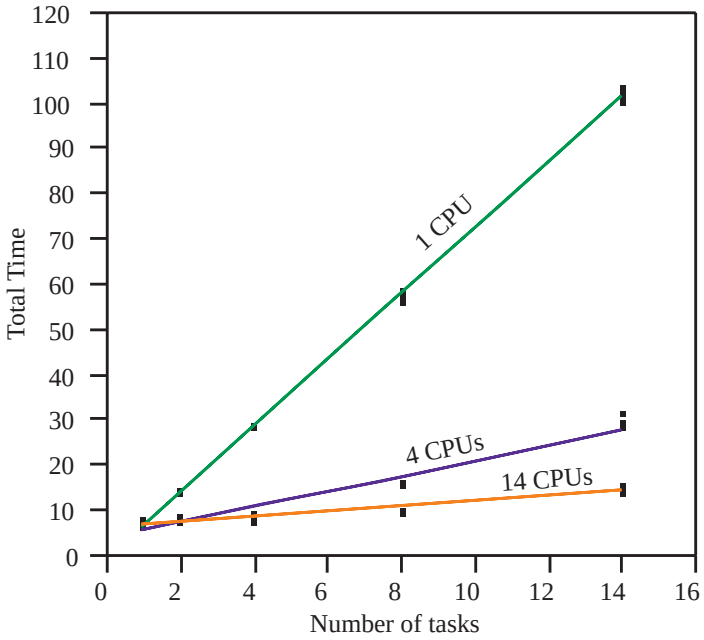

Figure 4. Threaded Execution Results

7.1.2. *Making Use of Unused Cycles in the JVM*

One premise of DECAF is that the architecture provides increased reliability by using unused CPU cycles to maximize throughput. This cannot work if all the actions are compute bound. However, in most cases there is an I/O component to the processing that will let other parts of the architecture run.

A test was run to verify that actions that are I/O bound will run in parallel even on a single processor. The following graph show the time to execute ten actions. The mix of the ten actions varies from 100% compute bound tasks (implying 0% I/O bound actions) to 100% I/O bound actions (implying 0% compute bound actions) Each I/O bound task runs for 5 seconds on its own and each compute bound task runs for about 2 seconds on its own. If things work as we need them to, 10 I/O bound tasks should run in 5 seconds plus some overhead for the context switching and architecture. Ten compute bound tasks should take around 20 seconds. When the type of tasks are mixed there should be a straight line correlation between time and the mix of jobs

to be done. Figure 5 shows the time to execute against the percentage of I/O bound tasks. This maps to the formula:

$$1.5 * C + 5 * G(I) + 1.23$$

where C is the number of compute bound tasks. 1.5(seconds) is the approximate running time of the compute bound task, G(I) is a function based on the number of I/O bound task ,

$$G(I) = \begin{cases} 1 & \text{if } I > 1 \\ 0 & \text{otherwise} \end{cases}$$

5 seconds is the time for the I/O bound task and 1.23 is the architecture and JVM overhead.

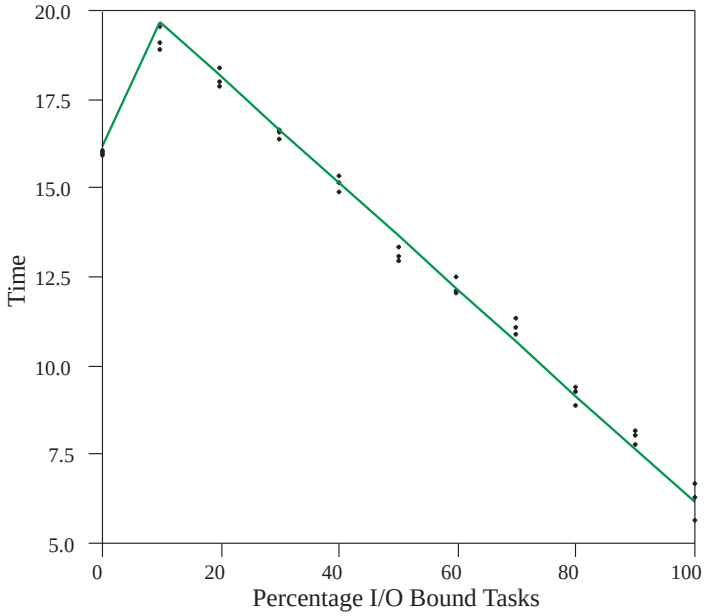

Figure 5. Task Type Results

7.2. THROUGHPUT AND NETWORK BEHAVIOR

Virtual Food Court (VFC)[31] is a small artificial economy (described later). VFC models diners, workers, and entrepreneurs. These economic entities are caricatures of the participants in transactions that take place within a simplified shopping mall food court.

VFC is a MAS and consists of seven basic agents plus some arbitrary number of diner agents and restaurant agents. Running VFC is interesting the context of our testing here because it can be scaled to 100's of diners

and spread out over many target machines to measure both scalability and network overhead.

The basic task unit for VFC is a meal. A diner enters the restaurant, negotiates a meal with a waiter. The restaurant makes decisions on whether to prepare the meal or buy it. The restaurant also must make decisions on the labor force (how many waiters to hire).

To eat one meal requires about 100 KQML messages to be exchanged with various agencies. It is also a very memory intensive application. Each agent must check on availability of employees, food and other resources. The test run varied the number of meals from two to one hundred. Tests were done where all of the VFC agent ran on one machine and the same test were done where the VFC agent were located on seven separate (single processor) machines on a local area network. The measurement of interest is the time it takes to service each diner with their meal.

The time to serve each meal increased as demand increased, just as it would take longer to get a meal in a crowded restaurant versus a not crowded restaurant. The time to get a meal with two diner was approximately 6 seconds per meal while the time per meal for twenty diners was approximately 25 seconds per meal.

An additional test was run. For the second test the input was identical and the components measured were the same but each agent in the community was located on a separate machine in a local network. The idea is to get some idea of network overhead as a function of the number of messages. The results indicate there is about 2% increase in time when run as a distributed multi agent system.

8. Evaluation: Case Studies

In order to show that the DECAF architecture works as a general purpose agent system, case studies of several projects using DECAF are presented as well as some of the support tools that have been developed for this project that we have not already discussed. Also, since a goal of DECAF is to be used as a research platform for the development of new component technologies, we also discuss some of these. There is no single test which demonstrates that DECAF is “correct” and “useful” just as no single task can verify the usefulness of the UNIX operating system. The usefulness of DECAF has been shown by using it as part of classes on multi agent systems and by actual development of systems. The following section are brief discussions of the more significant developments using DECAF.

8.1. GPGP

Generalized Partial Global Planning (GPGP) is a task structure centered approach to coordination [9]. The basic idea is that each agent constructs its local view of the structure and relationships of its intended tasks. This view is then augmented by information from other tasks and the local view will change dynamically over time. In particular, commitments are exchanged that result in new scheduling constraints. The result is a more coordinated behavior for all agents in the community. Currently a GPGP module is under development that will coordinate plans on the fly, as they are instantiated, if they include explicit references to non-local tasks ("clouds"). The precise coordination mechanism used can change from domain to domain, or from task to task. An example of a generic coordination mechanism supported by GPGP is a reservation request. If Agent A needs Agent B to carry out Task TB, then A can ask B to commit to a certain finish time for TB ahead of time. Given the general representations being used in DECAF, a large number of these coordination mechanisms can be pre-programmed and used without alteration by MAS application programmers.

8.2. Planning for a Smart Scheduler

The focus of planning in our system is on explicating the basic information flow relationships between tasks and other relationships that affect control flow decisions. Most control relationships are derivative of these more basic relationships. Final action selection, sequencing and timing are left up to the agent's local scheduler. Thus the planning process takes as input the agent's current set of goals and set of task structures and produces as output a new set of current task structures. The important constraint on the planning module is to guarantee at least one task for each goal until the goal is accomplished or until the goal is believed to be unachievable.

Special features of the planner include the ability to plan for preconditions and plan to achieve abstract predicate goals (instead of decomposition by task names). The planner also designs plans to allow runtime choices between branches to be made by an intelligent scheduler, based on user preferences that can change between plan time and runtime. This feature provides real time flexibility, since the scheduler can react to a dynamic environment by exploiting choice within a plan, rather than forcing the planner to do costly re-planning.

8.3. Information Extraction Agent

The main functions of an Information Extraction Agent (IEA) are [10]: Fulfilling requests from external sources in response to a *one shot query* (e.g.

"What is the price of IBM?"). Monitoring external sources for *periodic* information (e.g. "Give me the price of IBM every 30 minutes."). Monitoring sources for patterns, called *information monitoring* requests (e.g. "Notify me if the price of IBM goes below $50.")." These functions can be written in a general way so that the code can be shared for agents in any domain.

Since our IEA operates on the Web, the information gathered is from external information sources. The agent uses a set of *wrappers* and the wrapper induction algorithm STALKER [34], to extract relevant information from the web pages after beeing shown several marked-up examples. When the information is gathered it is stored in the local IEA "infobase" using Java wrappers on a PARKA [41] knowledgbase. This makes new IEA's easier to create, and forces the difficult parts of this problem back onto KB ontology creation, rather than producing tools to wrap web pages and dynamically answer queries.

8.4. MODELING WITH THE VIRTUAL FOOD COURT.

Virtual Food Court (VFC)[31] is a small artificial economy. VFC models diners, workers, and entrepreneurs. These economic entities are caricatures of the participants in transactions that take place within a simplified shopping mall food court. Although caricatures, the entities exhibit behaviors, chosen from a repertoire of self-interested behaviors, sufficient to allow VFC to contain a labor market, markets for food service equipment, and markets for food products. For example, accepting a contract to perform labor and forming an organization (i.e., offering the labor contract) are reciprocal events. Because both of these are voluntary actions, VFC models and explains both sides of the transaction simultaneously. VFC plans to extend our results to model organizational structures more complicated than a simple employment contract (while still, of course, basing the analysis on the need for there to be reciprocally voluntary contracts). Such models will be expanded to include aspects of governance and perhaps non-economic social forces as we explore the long term control and stability of such structures.

The initial configuration of VFC is shown in figure 6 as a collection of boxes and lines. Lines represent the KQML communications and the boxes are DECAF agents. Arrowheads indicate the direction of the initial message. In the DECAF paradigm, agents need to know of the Matchmaker in order to register their existence with it. Workers and Restaurants need to know of the existence of the Government because they are "required" to report to it.

8.5. GENEAGENT

GeneAgent[13, 14] is a bioinformatics-gathering system based on the RETSINA agent organizational concept of Interface Agents that work with humans, Information Extraction Agents that wrap various web resources, and

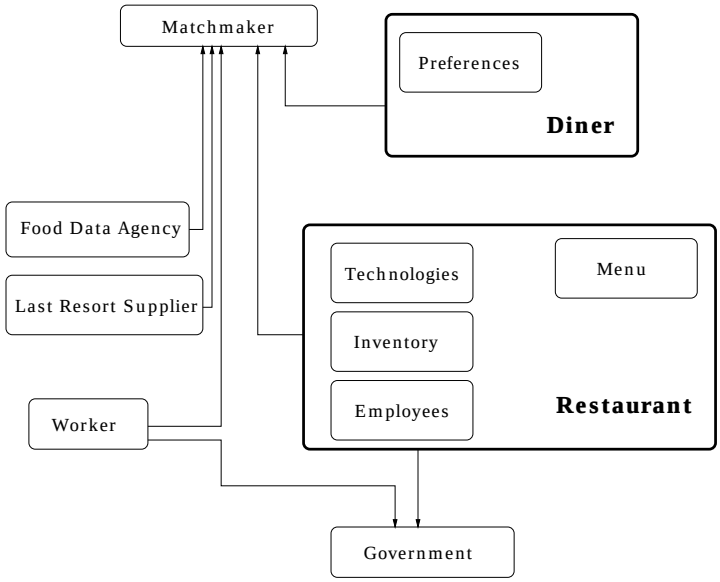

Figure 6. Virtual Food Court Architecture

Task Agents that include both middle-agents and domain task analysis agents. GeneAgent interfaces with a biologist via a browser and applets that use the DECAF Proxy Agent to communicate with the rest of the information gathering system. The Information Extraction Agent class has been used to build wrappers for several necessary Internet resources such as the NCBI BLAST servers that allow searching for protein sequences in GenBank; Protein Motif (sequence pattern) databases such as SwissProt, and local databases of organism-specific genetic sequences. Initial analysis agents provide services such as notification of new BLAST results and automated customized annotation of local genetic sequence information. Figure 7 shows the basic architecture of GeneAgent. An initial multi-agent test system for the search of the automated annotations of human and avian herpesvirus sequences is located at `http://udgenome.ags.udel.edu/herpes/`

9. Conclusions

In this paper we have discussed DECAF, a software toolkit for the design, development, and execution of "intelligent" agents to achieve solutions in complex software systems. DECAF is based on the premise that execution of the actions required to accomplish a task specified by an agent program is similar to a traditional operating system executing a sequence of user requests. Our vision for DECAF is to allow ourselves and other researchers a platform for experimentation in agent architectural components, but also to

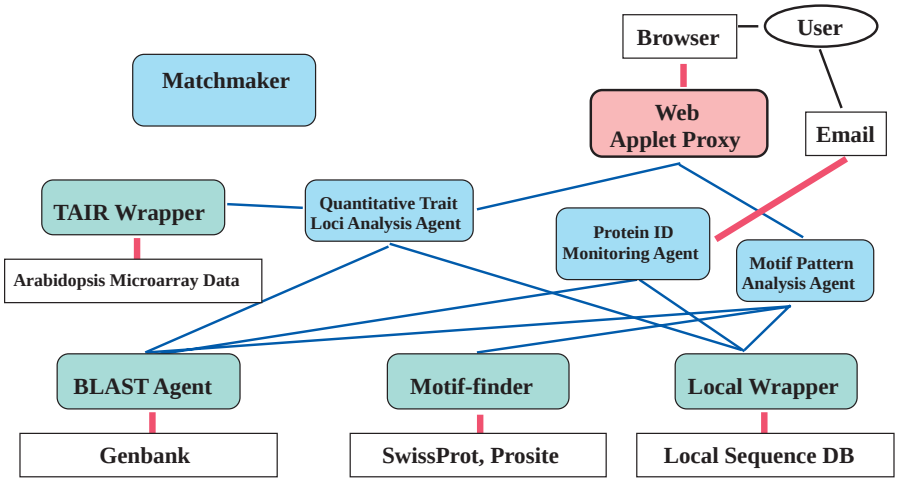

Figure 7. GeneAgent Architecture

support a level of multi-agent system engineering that is increasingly useful for students or application programmers that will allow them to demonstrate and use the power of multi-agent systems without having to design an agent from the ground up. Thus DECAF also focusses on programming agents, not designing internal architecture from APIs. It supports programming at the multi-agent level via the use of several fairly standard prebuilt middle-agents and generally useful agent classes such as a web information extraction agent. We have attempted to provide value-added features to the programmer such as taking care of common, important details such as ANS registration, robust behavior libraries to interact with middle-agents, etc. We also have an efficient architecture which makes much use of modern multi-threading and multi-processor machines. Finally, we provide support for persistent and flexible actions through the ability to program alternatives that are chosen dynamically at run-time, and plans with data-flow control based on multiple outcomes and input provisions allowing looping and other robustness-enhancing processes.

10. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants No. 9733004 and 9812764. We thank the reviewers and all of the people who have worked on the DECAF project in the present and past: J. Barbour, W. Chen, D. Cleaver, J. Eskelinen, T. Harvey, V. Hyatishin, M. Laukkanen, F. McGeary, D. McHugh, A. Rispoli, V. Windley.

References

1. Bach, M.: 1986, *Design of the UNIX Operating System*. Prentice-Hall.
2. Barrett, T., G. Coen, J. Hirsh, L. Obrst, J. Spering, and A. Trainer: 1997, 'MADEsmart: An Integrated Design Environment'. 1997 ASME Design for Manufacturing Symposium.
3. Bonasso, R., D. Kortenkamp, and T. Whitney: 1997, 'Using a Robot Control Architecture to Aoutomate Space Shuttle Operation'. In: *Proceding of the Ninth Conference on Innovative Appications of AI*.
4. Bond, A. H. and L. Gasser (eds.): 1988, *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann.
5. Brooks, R. A.: 1986, 'A Robust Layered Control System for a Mobile Robot'. *IEEE Journal of Robotics and Automation* **RA-2**(1), 14–23.
6. Cohen, P. R. and H. J. Levesque: 1990, 'Intention is Choice with Commitment'. *Artificial Intelligence* **42**(3), 213–261.
7. Decker, K. S.: 1987, 'Distributed Problem Solving: A Survey'. *IEEE Transactions on Systems, Man, and Cybernetics* **17**(5), 729–740.
8. Decker, K. S. and V. R. Lesser: 1993, 'Quantitative Modeling of Complex Computational Task Environments'. In: *Proceedings of the Eleventh National Conference on Artificial Intelligence*. Washington, pp. 217–224.
9. Decker, K. S. and V. R. Lesser: 1995, 'Designing a Family of Coordination Algorithms'. In: *Proceedings of the First International Conference on Multi-Agent Systems*. San Francisco, pp. 73–80. Longer version available as UMass CS-TR 94–14.
10. Decker, K. S., A. Pannu, K. Sycara, and M. Williamson: 1997a, 'Designing Behaviors for Information Agents'. In: *Proceedings of the 1st Intl. Conf. on Autonomous Agents*. Marina del Rey, pp. 404–413.
11. Decker, K. S. and K. Sycara: 1997, 'Intelligent Adaptive Information Agents'. *Journal of Intelligent Information Systems* **9**(3), 239–260.
12. Decker, K. S., K. Sycara, and M. Williamson: 1997b, 'Middle-Agents for the Internet'. In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*. Nagoya, Japan, pp. 578–583.
13. Decker, K. S., X. Zheng, C. Schmidt: 2001, 'A Multi-Agent System for Automated Genetic Annotation'. In: *Proceedings of the 5th Intl. Conf. on Autonomous Agents*. Montreal, pp. 433–440.
14. Decker, K. S., S. Khan, C. Schmidt, D. Michaud: 2001, 'Extending a Multi-Agent System for Genomic Annotation'. In: M. Klusch and F. Zambonelli (eds.): *Cooperative Information Agents V*. LNAI #2182, pp. 106–117, Springer 2001.
15. Durfee, E. and T. Montgomery: 1989, 'MICE: A Flexible Testbed for Intelligent Coordination Experiments'. In: *Proceedings of the Ninth Workshop on Distributed AI*. Rosario, Washington.
16. Erol, K., J. Hendler, and D. Nau: 1994, 'Semantics for Hierarchical Task Network Planning'. CS Technical Report TR-3239, University of Maryland.
17. Ferguson, I. A.: 1992, 'TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents'. Technical Report 273, Univeristy of Cambridge, Cambridge, UK.
18. Firby, R. J.: 1996, 'Task Networks for Controlling Continuous Processes'. Seattle, WA.
19. Fisher, M.: 1996, 'Introduction to Concurrent MetateM'.
20. Garvey, A., M. Humphrey, and V. Lesser: 1993, 'Task Interdependencies in Design-To-Time Real-Time Scheduling'. In: *Proceedings of the Eleventh National Conference on Artificial Intelligence*. Washington, pp. 580–585.
21. Gasser, L.: 1991, 'Social Conceptions of Knowledge and Action'. *Artificial Intelligence* **47**(1), 107–138.

22. Gasser, L.: 1998, 'Agent and Concurrent Objects'. *An interview by Jean-Pierre Briot in IEEE Concurrency*.
23. Graham, J. R. and K. S. Decker: 2000, 'Towards a Distributed, Environment-Centered Agent Framework'. In: N. Jennings and Y. Lespérance (eds.): *Intelligent Agents VI — Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, pp. 290-304, Lecture Notes in Artificial Intelligence 1757. Springer-Verlag, Berlin.
24. Graham, J. R. 'Real-time Scheduling in Multi-agent Systems'. PhD. Thesis, University of Delaware, 2001.
25. Grosz, B. and S. Kraus: 1993, 'Collaborative Plans for Group Activities'. In: *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. Chambéry, France.
26. Harvey, T. and K. Decker: 2001, 'Planning Ahead to provide Scheduler Choice'. In: *Proc. Workshop on Infrastructure for Scalable Multi-Agent Systems*. 5th International Conference on Autonomous Agents, pp. 105–113, May 2001.
27. Hayes-Roth, F., L. Erman, S. Fouse, J. Lark, and J. Davidson: 1988, 'ABE: A Cooperative Operating System and Development Environment'. In: A. H. Bond and L. Gasser (eds.): *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, pp. 457–490.
28. Horling, B., V. Lesser, R. Vincent, A. Bazzan, and P. Xuan: 1999, 'Diagnosis as an Integral Part of Multi-Agent Adaptability'. Tech Report CS-TR-99-03, UMass.
29. Kuokka, D., and L. Harada: 1995, 'On Using KQML for Matchmaking'. Proceedings of the First International Conference on Multi-Agent Systems, AAAI Press, pp. 239–245.
30. Malone, T. W. and K. Crowston: 1991, 'Toward an Interdisciplinary Theory of Coordination'. Center for Coordination Science Technical Report 120, MIT Sloan School of Management.
31. McGeary, F. and K. Decker: 2001, 'Modeling a Virtual Food Court Using DECAF'. In: S. Moss and P. Davidsson (eds.): *Multi-Agent Based Simulation*. LNAI #1979, pp. 68–81, Springer 2001.
32. McGeary, F. and K. Decker: 2001, 'DECAF Programming: Agents for Undergraduates'. In: *Proc. Workshop on Infrastructure for Scalable Multi-Agent Systems*. 5th International Conference on Autonomous Agents, pp. 53–60, May 2001.
33. Muller, J. and M. Pischel: 1994, 'Modelling interacting agents in dynamic environments'. In: *Proceedings of the eleventh Europen Conference on Artificial Intelligence*. pp. 709–713.
34. Muslea, I., S. Minton, and C. Knobloch: 1998, 'STALKER: Learning Expectation Rules for Semistructured Web-based Information SOurces'. Papers from the 1998 workshop on ai and information gathering. technical report ws-98-14, University of Southern California.
35. Charles J. Petrie: 1996, 'Agent-Based Engineering, the Web, and Intelligence'. IEEE Expert, December.
36. Rana, O. F. and K. Stout: 2000, 'What is Scalability in Multi-Agent Systems?'. In: *Proceedings of the Fourth Annual Conference on Autonomous Agents*. pp. 56–69.
37. Rao, A. and M. Georgeff: 1995, 'BDI Agents: From Theory to Practice'. In: *Proceedings of the First International Conference on Multi-Agent Systems*. San Francisco, pp. 312–319.
38. Rosenschein, J. S. and G. Zlotkin: 1994, *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers*. Cambridge, Mass.: MIT Press.
39. Russell, S. and E. Wefald: 1991, *Do the Right Thing: Studies in Limited Rationality*. Cambridge, MA: MIT Press.
40. Simmons, R.: 1996, 'Becoming Increasingly Reliable'.

41. Spector, L., J. Hendler, and M. P. Evett: 1990, 'Knowledge Representation in PARKA'. Technical Report CS-TR-2410, University of Maryland.
42. Sycara, K., K. S. Decker, A. Pannu, M. Williamson, and D. Zeng: 1996, 'Distributed Intelligent Agents'. *IEEE Expert* **11**(6), 36–46.
43. Wagner, T., A. Garvey, and V. Lesser: 1997, 'Complex Goal Criteria and its Application in Design-to-Criteria Scheduling'. In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence*. Providence.
44. Wagner, T., A. Garvey, and V. Lesser: 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.
45. Williamson, M., K. S. Decker, and K. Sycara: 1996a, 'Executing Decision-theoretic Plans in Multi-agent Environments'. In: *AAAI Fall Symposium on Plan Execution*. AAAI Report FS-96-01.
46. Williamson, M., K. S. Decker, and K. Sycara: 1996b, 'Unified Information and Control Flow in Hierarchical Task Networks'. In: *Proceedings of the AAAI-96 workshop on Theories of Planning, Action, and Control*.
47. Wooldridge, M. and N. Jennings: 1995, 'Intelligent Agents: Theory and Practice'. *The Knowledge Engineering Review* **10**(2), 115–152.