

# Scalability and Scheduling in an Agent Architecture \*

John R. Graham   Michael Mersic   Keith S. Decker

Department of Computer and Information Sciences  
102 Smith Hall, University of Delaware  
Newark, DE, 19716, USA

{graham, mersic, decker}@cis.udel.edu

**Abstract.** For a given agent goal or objective there will be many different possible task execution paths for achieving the goal. One method for achieving the goal is to algorithmically determine the “best” execution path execute that path. One question that arises with this strategy is what to do in the event a segment of the selected path fails. Then the agent architecture must select a new execution path and begin again. One potential solution to this problem is to start several execution paths, thereby increasing the chances for success. Questions of how much redundancy to use depend on factors such as the agent architecture, the amount of excess resources, the number of task interdependencies, the underlying hardware, and the network overhead. This paper will describe a methodology and results for building and evaluating task execution schedules of agent actions. In particular scalability, parallelism and the threaded nature of the individual agent architecture; the complexity of the scheduling algorithm itself; and the overhead of the architecture itself. Determining overhead is needed to be able to characterize the types of time constraints the architecture can respond to. Experiments using different task scheduling algorithms were utilized to develop an experimental platform for future research in agent task scheduling. The goals of the testing were to determine some basic parameters that will be used to determine the suitability of the architecture for certain types of work and to assist in development of efficient soft real-time task scheduling algorithms.

**Keywords:** Scheduling , Parallel Programming, agent task scheduling.

## 1 Introduction

In this paper we are conducting an exploratory study of the behavior of an agent architecture in response to various types of test profiles. In this case the exploration is to discover how the system will behave under particular conditions. One reason to do this is to provide a means for comparing competing systems. While raw performance measurements are valuable, their real value lies in the evidence they provide that leads to explanations for the performance differences. This paper will provide both measurements and explanations that will lead to better scheduling algorithms.

Specific goals for these experiments are:

---

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

- Provide a basis for the comparison of architectures.
- Determine the effectiveness of the threaded design of our architecture.
- Determine how the architecture will perform under heavy loads.
- Determine how the distributed nature of an agent task hierarchy affects performance.
- Compare several task scheduling algorithms <sup>1</sup> and then examine the results to determine why they worked the way they did.

The experiments here are conducted using the DECAF [5] agent architecture. The basic architecture is described, followed by the experimental setup and then the experimental results.

## 2 Overview of DECAF Architecture

DECAF (Distributed, Environment-Centered Agent Framework) is a toolkit which allows a well-defined software engineering approach to building multi-agent systems. The toolkit provides a stable platform to design, rapidly develop, and execute intelligent agents to achieve solutions in complex software systems. DECAF provides the necessary architectural services of a large-grained intelligent agent [3, 8]: communication, planning, scheduling, execution monitoring, coordination, and eventually learning and self-diagnosis [6]. This is essentially the internal “operating system” of a software agent, to which application programmers have strictly limited access.

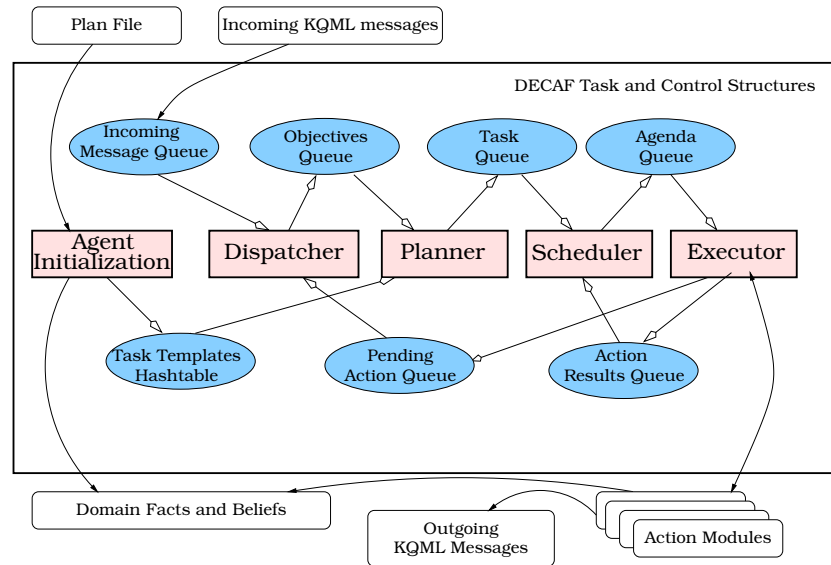
The control or programming of DECAF agents is provided via a GUI called the *Plan-Editor*. In the Plan-Editor, executable actions are treated as basic building blocks which can be chained together to achieve a larger more complex goal in the style of an HTN (hierarchical task network). This provides a software component-style programming interface with desirable properties such as component reuse (eventually, automated via the planner) and some design-time error-checking. The chaining of activities can involve traditional looping and if-then-else constructs. This part of DECAF is an extension of the RETSINA and TÆMS task structure frameworks [10, 2].

The goals of the architecture are to develop a modular platform suitable for our research activities, allow for rapid development of third-party domain agents, and provide a means to quickly develop complete multi-agent solutions using combinations of domain-specific agents and standard middle-agents [1] and to take advantage of the object oriented-features of the JAVA programming language.

The modular nature of the architecture will be utilized in several ways during the course of the experiments. First, each module of the architecture (the boxes in Figure 1 are established as separate threads in the Java language constructs. Further, the execution module establishes a separate thread for each executable action that is not dependent on another action for the task. It is anticipated that this highly parallel design will lead to large gains in computation speed. This speed of course is used meet “soft” real time goals and deadlines.

---

<sup>1</sup> In this paper “task scheduling” refers to the act of allocation of processor resources to agent actions and not to domain-level or distributed actions.



**Fig. 1.** DECAF Architecture Overview

Secondly, the scheduling module itself will be replaced with another scheduling module to compare results. The scheduling can be tested in isolation from the remainder of the architecture. Lastly, the Dispatcher module will be tested by setting up a highly distributed and very network intensive test. Again the parallel nature of the code should reveal that an I/O intensive operation such as network communications can be done with little impact on the computation problems.

DECAF distinguishes itself from many other agent toolkits by shifting the focus away from the underlying components of agent building such as socket creation, message formatting, and the details of agent communication. In this sense DECAF provides a new programming paradigm: Instead of writing lines of code that include system calls to a native operating system (such as *read()* or *socket()*) DECAF provides an environment that allows the basic building block of agent programming to be an agent action. Conceptually, we think of DECAF as an agent operating system. Code within that action can make calls to the DECAF framework to send messages, search for other agents or implement a formally specified coordination protocol. This interface to the framework is a strictly limited set of utilities that remove as much as possible the need to understand the underlying structures. Thus, the programmer does not need to understand JAVA network programming to send a message, learn JAVA database functions to attach to the internal knowledge base of the framework, or implement standard coordination mechanisms.

### 3 The Experimental Study

#### 3.1 Defining Scheduling

For a given task, there will potentially be many different methods for accomplishing the task. The scheduling “problem” is the selection from a set of possible execution paths, one path that accomplishes the agent goal with idea of a “best” execution path. If, however, we can attach to each execution path some kind of estimation of the performance of that path, AND the user can provide an idea of what the performance objectives are; then the agent architecture can reason about the concept of a “best” execution path. One of the goals of the experiments described here is to determine what parameters might be used for reasoning about scheduling. In order to do that, the modular design of DECAF allows for the implementation and testing of any number of scheduling algorithms. Since the inputs and output to the scheduling module are well defined it is relatively simple matter to change algorithms and run experiments.

DECAF operates by reading a *plan* file which contains a list of tasks that this instantiation of the architecture is capable of performing. A plan file is an ASCII representation of a Hierarchical Task Network (HTN) that details the actions and sequences to complete a task.

The DECAF Plan-Editor annotates each action with notes on performance and use which are then used internally by DECAF to provide real-time local task scheduling services.

- A **performance profile** is a vector of characteristics

$$\vec{C} = \langle C_1, C_2, \dots, C_n \rangle$$

assigned to DECAF **actions**. These characteristics relate to the performance of the action itself and not to the result produced by the action. Examples of characteristics may be cost, quality and duration.

DECAF will select a set of actions to be executed that will accomplish the goal. For a given plan file, there will be many such sets. Each set of actions is evaluated and assigned a value as follows.

- Each **task** uses a **Characteristic Accumulation Function** (CAF) which determines how the vector of performance characteristics related to the achievement of a task is related to the achievement of the component subtasks (or actions). The CAF takes as input a list of characteristic vectors, and produces a vector of characteristics for the task.

$$\vec{CAF}(\vec{C}_1, \vec{C}_2, \dots, \vec{C}_n) \rightarrow \vec{C}_{CAF}$$

- An **objective utility function** ( $\mathcal{U}$ ) is an assignment by the user of some idea of “goodness” of the actions that make up the task.  $\mathcal{U}$  takes as input the characteristic vector produced by CAF and produces a value that quantifies the “goodness” of the overall task.

The result is a set of values assigned to sets of actions. The set of actions with the best  $\mathcal{U}$  is the set that will be chosen to be executed. 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.

There is a considerable amount of computational complexity involved in determining the “best” execution path. In the event of a sub-task failure, that complexity must be repeated in order to determine the next course of action. Our plan of attack has two components:

- Provide faster service by making the task model simpler.
- Provide better reliability by selecting several sets of actions to avoid the computational overhead by

The focus of the experimental results done here are on the Scheduling module. 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 agent action 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.

Initially, this module of DECAF is based on the design-to-time (DTT) and design-to-criteria (DTC) scheduling work at the University of Massachusetts [4, 9]. For comparison purposes, there will be two other (non-reasoning) modules compared to DTC.

### 3.2 Experimental Goals and Setup

DECAF operates by reading a *plan file* which contains a list of tasks that this instantiation of the architecture is capable of performing. A plan file is an ASCII representation of a Hierarchical Task Network (HTN) that details the actions and sequences to complete a task. The actual syntax of the plan file is an extension of the RETSINA and TÆMS structure detailed in [10, 2]. In broad terms, a plan file is *tree*<sup>2</sup>. The plan defines execution paths along the various branches of the tree and the critical measurement of complexity of the plan is the number of actions (represented as tree leaves) to be executed.

The test scenario will be a set of plan files and a set of scheduling modules. The following sections describe the setup for each of areas of exploration.

---

<sup>2</sup> “Tree” is not quite a totally accurate term for an HTN plan, but for purposes here it will serve well.

**Testing Scheduling** A DECAF agent is programmed via a plan file. Each plan file will have a defined *Complexity Rating*, CR. The CR is a quantification of the inherent difficulty analyzing a plan and coming up with preferred schedules. In general, there is no convenient way to characterize CR, meaning there is no simple mathematical formula in terms of breadth and depth of the tree. This is because a branch of the tree may be taken or not taken based on the CAF defined earlier. Algorithmically, the CR is determined by a recursive traversal of the HTN which counts the the total number of possible paths that can be taken to accomplish a task taking into account the CAF.

The premise of this test is that the model of a task defined by DECAF is less complex than a TÆMS model. This means for the same task there will be fewer paths to explore for possible schedules. One objective of the scheduling test will be to determine a relation between and execution time. Clearly the paths to be analyzed the more time it will take. In comparing DTC to the DECAF scheduler, there is not a one-to-one correspondence. The reason is that TÆMS Quality Accumulation Function (QAF) (roughly the equivalent of the DECAF CAF) allows for more types of semantics than DECAF and while DECAF has only one type of enabling relationship TÆMS has approximately ten. It should be noted that all constructs of TÆMS are achievable in DECAF via macros or function mapping. In this sense, DECAF is a more RISC architecture while TÆMS follows a CISC architecture.

The basic test then is to compare the time to compute the schedules for DTC and DECAF plan files that have the same CR. At this time we do not have such results. The problem is that one plan file in DECAF will have a much lower CR than the same plan file applied to DTC. We are currently in the process of developing plans that have similar CR and then measuring the time to compute a schedule.

As an example, we measured the CR for BIG (Bounded Information Gathering) [7]. BIG was run through DTC and through DECAF. BIG is a sophisticated, web-based information gathering agent that recommends software packages. BIG plans, locates and processes free-format WWW documents via natural language processing and other text extraction techniques. BIG consists of 26 methods. In TÆMS (DTC), this results in greater than 67 million paths. To evaluate the paths and come up with the best 15 schedules took DTC 6 seconds. The same plan run though DECAF was evaluated in 2 seconds and the best plan had the same overall utility as the DTC evaluation.

**Testing Scalability** 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. For these tests the  $\mathcal{CR}$  is not so much an issue as the total number of actions involved in the system. There should be a relation between the number of agents and the execution time.

Additionally, An alternative execution module was tested. 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 will give 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.

## 4 Experimental Results

### 4.1 Multi Processors versus Single Processor

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 2 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.

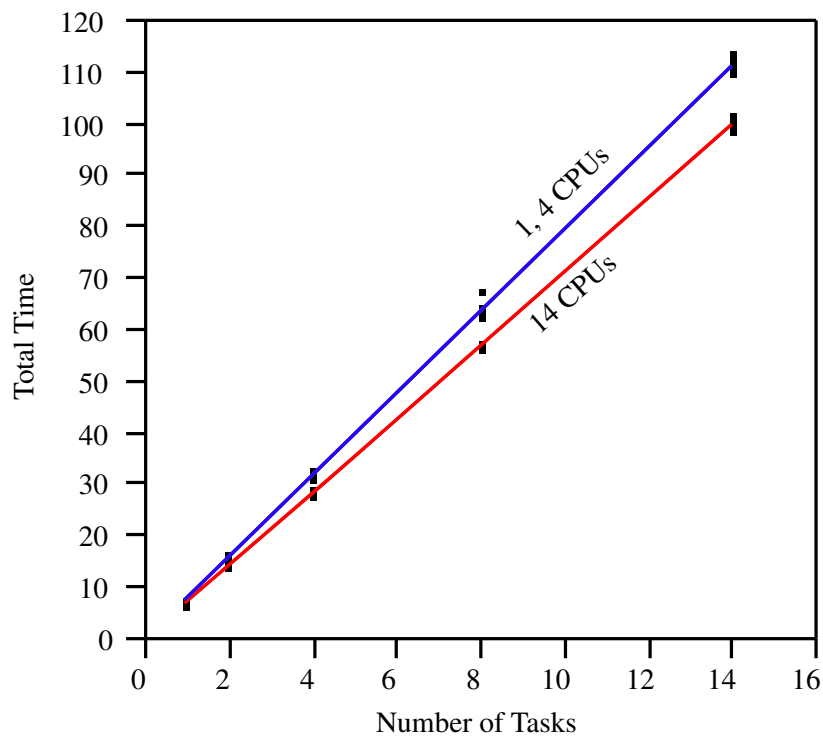
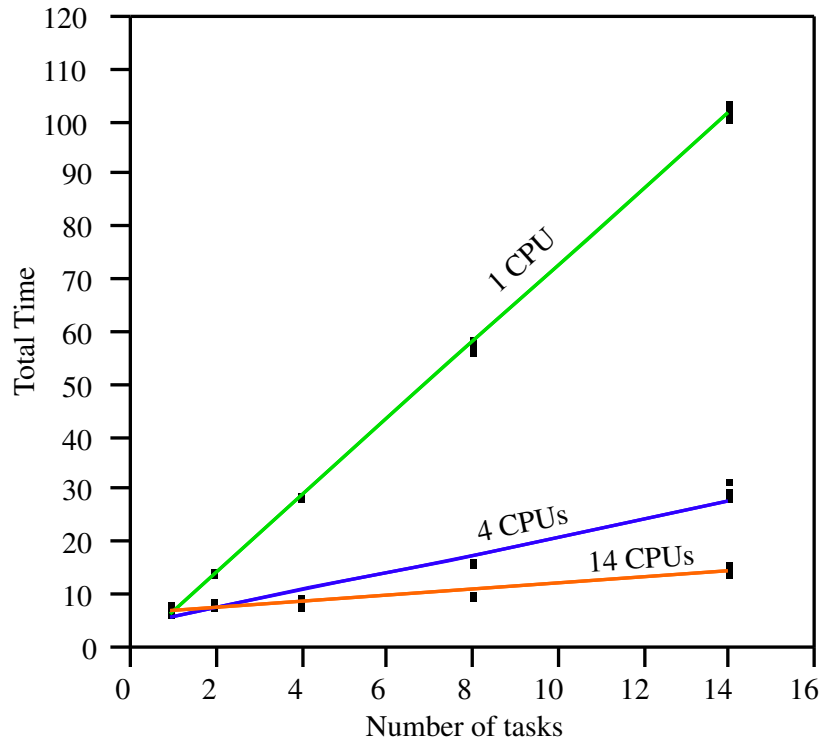


Fig. 2. Non-Threaded Execution Results

## 4.2 Threaded versus Non-threaded Execution

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 3 you can see that if there is only one processor, there is nothing to be gained by multi-threading. The difference between the values shown in Figure 2 and Figure 3 is the result of threading and shows that the JVM works in the expected manner.



**Fig. 3.** Threaded Execution Results

In terms of our ultimate goal of running several execution schedules in parallel this is significant. The idea is to improve reliability through redundancy. These tests show that spare CPU cycles will be available for such redundancy only if an action is not 100% compute bound as these tasks were. At this time we are developing test that show such increased reliability can be achieved.

## 4.3 Feasibility

The basic premise of the manner that DECAF will provide increased reliability is that unused CPU cycles can be used to run backup plans in the case of the failure of the

primary schedule selection. This of course cannot work if the actions to be computed are compute bound. 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 4 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 ,

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

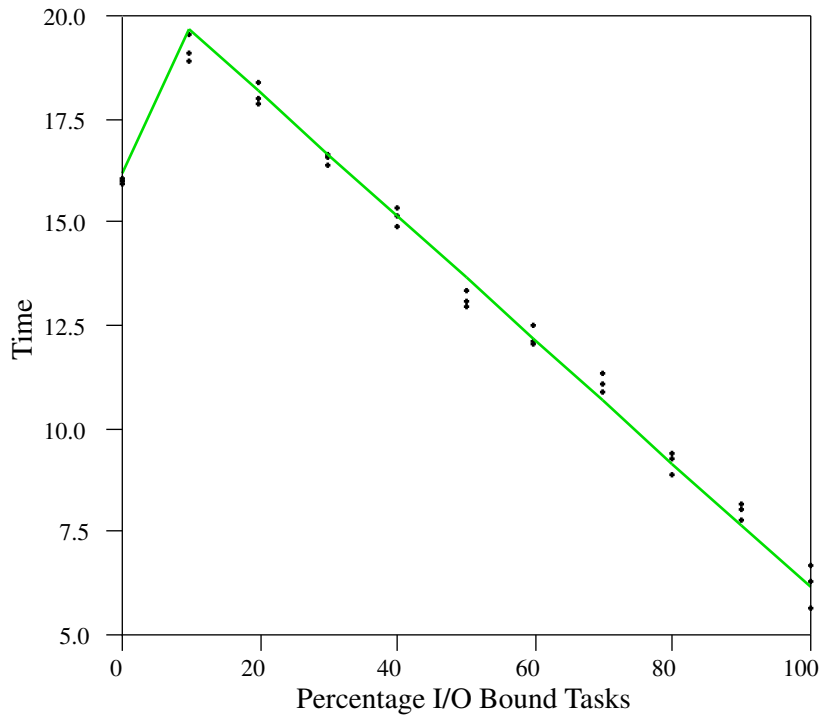
5 seconds is the time for the I/O bound task and 1.23 is the architecture and JVM overhead.

#### 4.4 Network Intensive Execution and Agent Scalability

Virtual Food Court (VFC) 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. VFC is a MAS and consists of seven basic agents plus some 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 machine 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. To eat one meal requires about 100 KQML message to be exchanged with various agencies. It is also a very memory intensive application. Each agent must checks 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 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. For example, 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. This



**Fig. 4.** Task Type Results

means that the number of messages sent to accomplish serving the meals also increased but the difference between the local execution and the distributed execution was constant at about %2. A further test would be to run the agents on completely different networks, but that is not feasible in our domain.

## 5 Conclusions and future work

DECAF was first presented at ATAL-99. At this time, DECAF is a much more mature product and several complex agent systems have been developed using DECAF. Two academic classes have used it for classwork, the Virtual Food Court is being used to model economic behavior, a proxy agent for interface with browsers, a web page learning agent and a PARKA database agent have been developed using DECAF.

The major goals of our testing were to:

- Provide a basis for the comparison of architectures.
- Determine the effectiveness of the threaded design of our architecture.
- Determine how the architecture will perform under heavy loads.
- Determine how the distributed nature of an agent task hierarchy affects performance.

- Compare several task scheduling algorithms.

The obvious improvement in performance due to threading is clear. However, some improvement in the internal mechanisms of DECAF may be needed to fully take advantage of a multiprocessor system. Also the changes in networked agents compared to locally executed agent systems needs to be examined.

One major goal of the testing was to develop a platform for the types of tests and measurements that need to be used in evaluating agent performance. In the course of this testing a test generator was developed which proved invaluable for creating various types of tests. The test generator allows for agent task structures of different complexity to be evaluated. Future work will focus on the development of a more complete set of benchmarks for evaluation of DECAF as well as other agent architectures.

Another goal of the testing was to establish a baseline for comparison of scheduling algorithms. This has been achieved and in the process we have established the validity of the structure of the DECAF model. It proved to be a relatively simple task to exchange scheduling algorithms since the interface to the module is well defined.

## 6 Acknowledgments

Primary development of DECAF was done by John Graham with extensive improvement by Mike Mersic. Development of the Virtual Food Court was by Foster McGeary. The test generator was developed by David Cleaver.

## References

1. K. S. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 578–583, Nagoya, Japan, August 1997.
2. Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 217–224, Washington, July 1993.
3. Keith S. Decker and Katia Sycara. Intelligent adaptive information agents. *Journal of Intelligent Information Systems*, 9(3):239–260, 1997.
4. Alan Garvey, Marty Humphrey, and Victor Lesser. Task interdependencies in design-to-time real-time scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 580–585, Washington, July 1993.
5. John R. Graham and Keith S. Decker. Towards a distributed, environment-centered agent framework. In N.R. Jennings and Y. Lespérance, editors, *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, 2000. In this volume.
6. B. Horling, V. Lesser, R. Vincent, A. Bazzan, and P. Xuan. Diagnosis as an integral part of multi-agent adaptability. Tech Report CS-TR-99-03, UMass, 1999.
7. Victor Lesser, Bryan Horling, Frank Klassner, Anita Raja, Thomas Wagner, and Shelly Zhang. Big: A resource-bounded information gatehring and decision support agent. *IEEE Internet Computing*, March/April 2000. A version also available as UMASS CS TR-98-52.

8. K. Sycara, K. S. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6):36–46, December 1996.
9. T. Wagner, A. Garvey, and V. Lesser. Complex goal criteria and its application in design-to-criteria scheduling. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Providence, July 1997.
10. M. Williamson, K. S. Decker, and K. Sycara. Unified information and control flow in hierarchical task networks. In *Proceedings of the AAAI-96 workshop on Theories of Planning, Action, and Control*, 1996.