Multitasking Strategies in Support of a Knowledge-Based Operator Companion

A.S. Mahmoud*, Wm. J. Garland*, W.F.S. Poehlman**
*Department of Engineering Physics
**Department of Computer Science and Systems

ABSTRACT

Today's operator, of nuclear-electric generating stations and chemical process plants functions in a data and information rich environment. In an effort to aid such operators, the structure of a real-time knowledge-based advisor is herein investigated for the central sampling system at Pt. Lepreau Generating Station. Human expertise (inherently symbolic or pattern recognition based) must be married to the speed and pervasiveness of computerized data acquisition and analysis which reside more in the domain of numeric processing. The schema advanced in this proposal achieves functional and temporal abstraction by using a tiered decomposition of the tasks, linked by real-time asynchronous agents acting through a blackboard.

An on-line behaviour can be achieved by developing a specific instance in the small of functional and temporal abstraction for a real-time system using the anthropomorphic approach of a blackboard partitioned along the lines of manager - supervisor - technician.

Any Overhead encountered in conforming an engineering domain problem to the suggested configuration is out-balanced by many gained advantages such as parallel pursuit of the primitive solution segments and the capability of changing focus of the system.

DESQView, a multitasking DOS shell provided a test platform. A prototype system was developed to test message passing techniques and problem decomposability.

1 INTRODUCTION

Operating plants such as nuclear-electric generating stations and chemical process plants are typically large and complex, incorporating remote and indirect sensors. Today's operator of such facilities function in a data and information rich environment. Mechanisms are needed to help the operator assess and assimilate this information to assist in the decision making process [2]. In this regard, tedious or standardized procedures that are normally performed by the expert operator can easily be relegated to an intelligent, computerized agent. Under such a scheme the operator is then better able to focus on higher level duties.

There is an expressed need for an intelligent advisor for central sampling. New Brunswick Electric Power Commission personnel at Pt. Lepreau Generating Station have provided the investigators with the operating manual for the Central Sampling System for Secondary Side Chemistry. This manual describes, among other things, how to detect and react to two separate but related concerns: bad secondary side chemistry (which has a very detrimental effect on the steam generators) and turbine condenser tube failure (a mechanical event that permits the ingestion of sea water into the feedwater, causing bad chemistry and other problems). Bad chemistry can result from normal system transients and is not a concern if the symptoms do not persist. If the problem persists, then the operator is to respond by derating, holding power, shutdown, etc. as appropriate to the details of the problem, irrespective of the cause of the chemistry problem. Leak detection initiates a separate response, also irrespective of the chemistry problem. Timely repair is required to minimize subsequent damage, subject to safety and cost considerations.

Much "real-time" expert system activity has focused on the electric power plant. Such a plant process exhibits typical man-machine interaction of modern industry and has many facets for computerized study. They include diagnosticians [3] with novel learning techniques [4], test-beds for numeric/symbolic coupled systems and some using progressive reasoning techniques [5]. Most papers point to bottlenecks with respect to uncoordinated numeric/symbolic processing of heterogeneous systems (we propose a blackboard system to accommodate intercommunication processes), concurrency (we propose an assembly of autonomous agents, some algorithmic and some intelligent, communicating asynchronously), and lack real-time capability [6] (we propose a tiered structure for architectural configurations where temporal and functional abstraction allow performance enhancements over traditional composite systems).

All the previously mentioned systems do not deal with a continuous stream of data. Not only must plant process analysis keep pace with a small snapshot of information flow, as is evident above, but real systems must analyze process behaviours with time series data which implies that the intelligent system be responsive to information updates as well as to dynamically be capable of

changing thinking directions due to this new data. As in the instance of robotic navigation which is just in its infancy, segmented thinking along a generalized to specialized goal seeking approach must be accommodated [7]. To our own knowledge, no system has been developed for plant process operation that exhibits these characteristics today. Even the comprehensive work undertaken by Ionescu and Triff [8] recognize this behaviour as a key concept but provide only an incomplete knowledge base to explore feasibility in their Operator Adviser Expert for a nuclear power plant.

Under such a situation, real-time response by the knowledge-based system is mandatory. We believe that on-line behaviour can be achieved by incorporating a tri-level configuration where, anthropomorphically:

(1) a manager addresses goals
(2) a supervisor carries out objectives, and
(3) a technician is concerned with direct implementation.

From top to bottom, this provides temporal and functional abstraction, proceeding from slow to fast response, and from the general to the specific. This is related to a tiered level of architectures. At the top level, knowledge-based or symbolic processors interconnect via an intelligent blackboard agent and at lower levels, numeric processors react in a traditional high performance manner.

The knowledge-based system above must be constrained to operate on a hardware platform at the PC level, but the software environment is unconstrained. Mainly two options were considered: UNIX and DOS. UNIX is proving inappropriate because this operating system isolates the user from the hardware too well, making it very difficult, if not impossible, to obtain multilevel interrupts. The focus of UNIX is to provide an efficient interactive multiuser interface for the computer rather than providing responses to external events through an interruptable environment. In the early experience with UNIX, the system proved to be unwieldy when implementing message passing mechanisms. This motivated the use of the simpler QuarterDeck DESQView program (a DOS multitasking shell). DESQView provides a cheap, easy to implement multitasking environment. This shell now provides convenient testing of different configurations of the anthropomorphic model. Lessons learned will guide future work.

2 DESCRIPTION OF DESQVIEW

2.1 DESQView a DOS-Multitasking Environment

DESQView runs as a shell over DOS allowing compatibility with the vast majority of DOS applications. In addition to multitasking, DESQView offers a variety of service calls known as *Application Program Interface* (*API*). The API services support:

1. Character-based windowing.
2. Multitasking.
3. Pointer devices (such as the mouse).
4. Timers objects.
5. Panels.
6. Interprocess communications.

2.2 DESQView Tasks and Processes

A task in DESQView is a single-execution thread. A process is a virtual machine environment that appears to a program to be a whole separate computer. Thus a single tasked application consists of a single task (one execution thread) within a single process (the application's code, data and window memory). A process can spawn a task using the function call *tsk_new()*. It returns a task handler which is a unique unsigned integer that identifies the task. Invoking *tsk_new()* is very similar to calling a function, except that control returns to the caller before the subtask completes. Because a task is defined within the parent process it has access to global variables and structures.

Just as a task can spawn a subtask in the same process, so can a task spawn a subtask in a different process; that is, one program can execute another program. The API call *app_start()* is used. This function returns a task handler but in contrast to *tsk_new()* the created subtask is completely independent and has no access to code and data of the spawning task. They may communicate via interprocess connections, using mailboxes.

2.3 Intertask Communication and Messages

For tasks within the same process, global variables can be used for communication. To avoid data collision situations, an application may disable DESQView's scheduling algorithm or use semaphores.

For tasks in different processes, DESQView provides mailbox objects which, can be used for communication. The sender task writes to the receivers mailbox a string of characters using the API call *mal_write()*. The receiver uses *mal_read()* to read his mailbox. There are two types of message passing techniques:

1. Passing by value: where the entire message is copied to the destination, and
2. Passing by reference: only a pointer to the message structure is passed to the receiver.

3 DESCRIPTION OF BLACKBOARD MODULES UNDER DESQVIEW

The proposed advisor is fashioned anthropomorphically after the typical management structure of:

1. Manager (Planner) level:
   The highest level of abstraction. It is only concerned with the questions like: What is the problem?. Is it a leakage problem?.
2. Supervisor (Mediator) level:
   It is concerned by how to solve or detect the problem.
3. Technician (Grunter) level:
   This level may contain little conventional module(s) that carry out instructions from supervisor or do data acquisition to support or negate a hypothesis.

This is not to say that the manager does not consider HOW or has nothing to DO, or that the other levels are equally single minded. The levels merely state what the primary task of the various modules are. Thus, in essence, the Central Sampling Advisor, as currently envisioned, is composed of separate modules modelled along the lines of the above three levels.

The Manager decides what issue to look at. In this case: "Is there a leak? If so, how big?" and "Is there a chemistry problem? If so, what action is required?". The manager poses the questions to the appropriate supervisors if they exist and are not busy (on a priority interrupt basis). If a specific supervisor is not available, then the questions can be posted to all supervisors in the hope that one can respond. The Manager monitors its incoming mail and reacts.

The Chemistry Supervisor responds to incoming requests (its in-basket) and reacts by posing two questions of its own: "Is the Condensate Polisher in service?" and "Does a transient condition exist in the plant?". The supervisor knows which technician can supply the answers to these questions. When the replies are received, the supervisor invokes one of four technicians to monitor a subset of the plant and chemistry data and to perform a fixed set of rules to determine the course of action. The Supervisor then posts messages to the Manager.

The Technician responds to incoming requests in a fixed manner. It needs to gather the required data and fire the rules of its domain in a forward chaining manner. The only difficult aspect is the time varying nature of the data. Discussion with plant personnel revealed that, in general, it is not necessary to retain past history. It is sufficient to look at the current data and apply the domain rules. At the time of rule firing, flags are set to denote significant events. Thus, for leak detection, it is sufficient to record that a leak was detected and at what time it was detected. Subsequent analysis at a later time will thus have the needed knowledge of the event.

To study the details of communications among various parts of the final system, a simpler system was constructed. Although a one to one map exists between the anthropomorphic model and the simpler system, the modules in the simpler system perform a subset of the tasks proposed in the model. The

system was built to run under QuarterDeck DESQView multitasking shell. In this prototype system the problem is to coordinate the activities of different types of agents, possibly more than one copy of each, in a way to guarantee concurrency and consistency of the actions. The prototype system consists of three types of agents. One type of agent creates files of random numbers while agents of a second type access these files to compute the average. The third type of agents access the data files to compute the variance, using the previously computed average. All agents report back with the results of their actions to the module whose responsibility is to maintain the board. This module is the Board Handler. Its job is to guarantee the consistency of the results and that only one action is being done on a data file at a time.

The current version is implemented using API service routines. In particular the RAM mailboxes, the event interrupts, and window control facilities were used to improve the message passing mechanism as well as the user interface. An earlier version used DOS files as a communication media. The architecture of the current prototype is shown in Figure 3.1. It consists of three types of modules at three different levels:

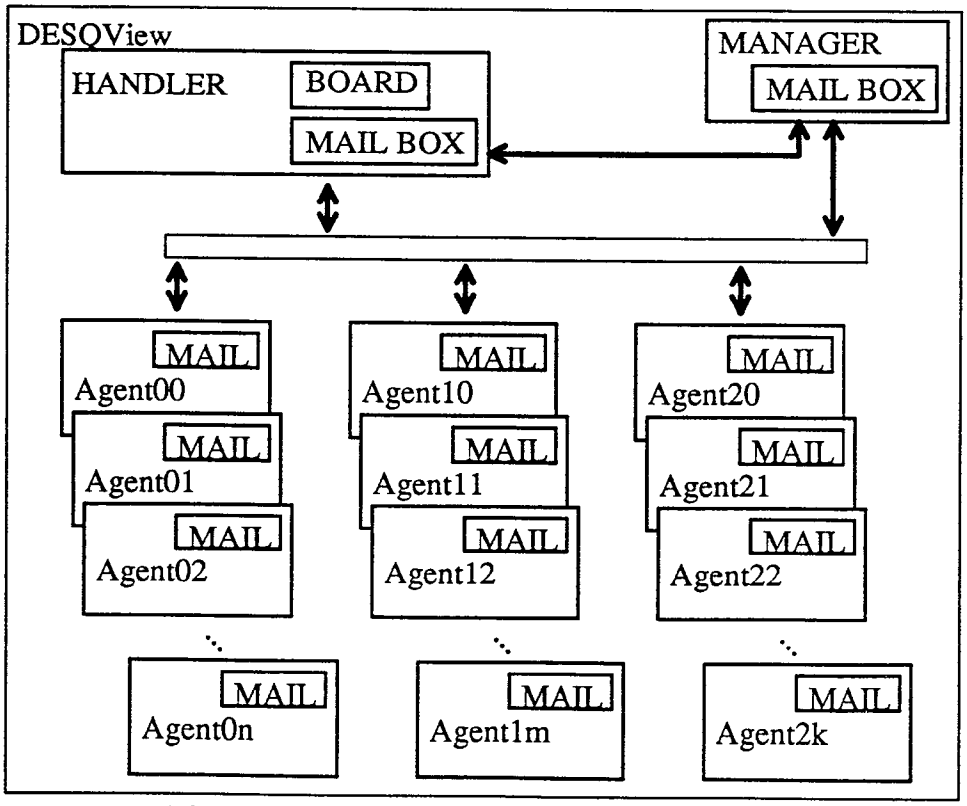

Figure 3.1: Architecture of Second Prototype

1. MANAGER level: A manager module is responsible for spawning other modules and sends administrative messages to these other modules.

2. BOARD HANDLER level: A module whose job is to maintain, and update the board. It also assigns tasks to agents.
3. Agents level: The primary task of agent modules is to carry out instructions received from Board Handler and report back with results.

3.1 The MANAGER Module

This module can generate the other modules one at a time by choosing the corresponding field on the window. It can terminate or control the window of other modules as well. This is done by sending messages to the module of concern. More functionality will be added to this module when the real problem is implemented.

In the current version, the MANAGER responds mainly to either user input, through mouse or keyboard, or incoming mail messages from other modules. Aside from these events, it is interrupted by two timer objects. The first timer object interrupts each second to update the clock displayed in a window. The second one checks the status of the overall system periodically. If there is no events to look after, the MANAGER releases the time slice allocated to it by DESQView's multitasking scheduling algorithm.

The expert system (which is being developed and tested separately) can finally reside in this module (or be partitioned between MANAGER and BOARD HANDLER to avoid bottlenecks). The MANAGER can poll the BOARD HANDLER for a report or even the complete board structure to be examined by the expert system. Thus, according to the evaluation of the board, new types of actions can be taken. For instance, messages could be sent to the BOARD HANDLER to focus on a certain activity.

3.2 The BOARD HANDLER

BOARD HANDLER is the module responsible for maintaining and updating the board structure. It is also responsible for assigning the tasks to available agents in a way that optimizes the CPU usage and guarantees no collisions among agents occur.

3.2.1 BOARD HANDLER Data Structures Two main data structures are the Board list and the Agent Lists. They are both manipulated and accessed only by this module. The Board structure holds domain information on the status of progress and on all the partial solution segments. The Board is tailored to the current problem, and a snapshot of it is shown Figure 3.2.

Keeping track of available agents is done through the agent lists, a data structure shown in Figure 3.3. Since there are three types of agents, there are three agent lists and an agent list is composed of agent nodes. The following describes the fields in an agent node:

File No.	Creation Action			Average Action				Variance Action				Agenda
	Itr	Chk	PID	Itr	Val	Chk	PID	Itr	Val	Chk	PID	
0	10	Y	agn01	9	51	N	agn11	9	720	N	agn22	C
1	11	Y	agn02	10	50	N	agn11	10	750	N	agn22	C
2	11	N	agn03	11	50	N	agn11	10	754	Y	agn22	V
3	11	N	agn01	11	52	N	agn11	10	741	Y	agn21	V
4	11	N	agn02	11	49	N	agn12	10	745	Y	agn21	V
5	9	N	agn02	8	53	Y	agn12	8	755	N	agn22	A
6	9	Y	agn01	8	51	N	agn12	8	753	N	agn22	C
7	13	Y	agn01	12	47	N	agn12	12	743	N	agn22	C
8	12	N	agn03	11	50	Y	agn13	11	747	N	agn21	A
9	10	N	agn03	9	52	Y	agn13	9	755	N	agn23	A

Figure 3.2: The Board in BOARD HANDLER module.

- Agent ID: the name of the agent.
- Mail Key: an unsigned long integer. It is the handler of the agent's mailbox.
- Active Flag: this flag is set to one as long as the agent is working and is participating in the activities. When it is excluded (not to be assigned any more tasks) this flag is set to zero.
- Total Load: an integer value indicating how many data files the agent must work on.
- Exclusion Timer: once the agent is assigned a job, this timer is set to the system clock reading. This timer is checked against the system clock periodically. Since an agent is expected to finish the assigned job and report back within a fixed time window, an agent is marked inactive and excluded if a preset time period has elapsed without the agent reporting to BOARD HANDLER.
- BigHead: head pointer to a list of nodes each called BigNode. Each BigNode node corresponds to a certain time frame (or stamp) and is the head of a load list.
- Next: it is a pointer to the next agent node.

The first interaction between the BOARD HANDLER and an agent is that the agent declares itself by sending a declaration message "*DEC*". This message contains the necessary information to initialize an agent node in the agent list structure.

3.2.2 BOARD HANDLER's Main Loop This module starts by initializing the Board structure as well as some global variables. Then it draws its window and records its name on its mailbox. Recording the name is necessary to prevent multiple copies of this module from existing at the same time. Finally the module sits in a loop waiting for events to occur. It expects three types of events: user input, an incoming mail message, or a timer interrupt. If there are

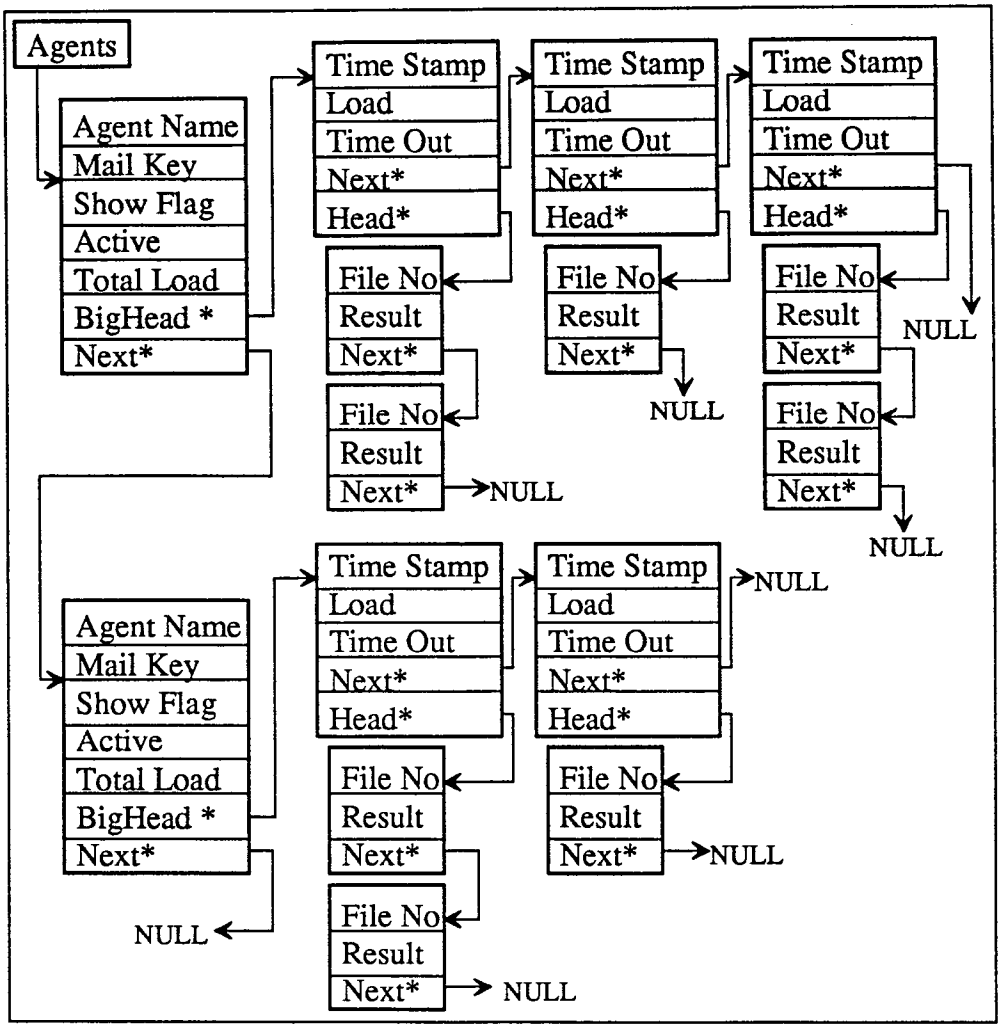

Figure 3.3: Agent List data structure.

no events to serve, it releases the time slice. Figure 3.4 depicts a flow chart of the main loop. A global variable, *TimeStamp*, is incremented each iteration of the loop.

When a mail event occurs, the BOARD HANDLER calls the *ProcessMailEvent* function. In this function the following is done:

1. Incoming messages are buffered in a message list to be processed.
2. Messages are processed according to their type. This processing may result in updating the Board structure as well as the Agent lists.
3. The board is evaluated and the required actions are recorded in an agenda.
4. Tasks in the agenda are distributed to available active agents by writing messages to their mailboxes.

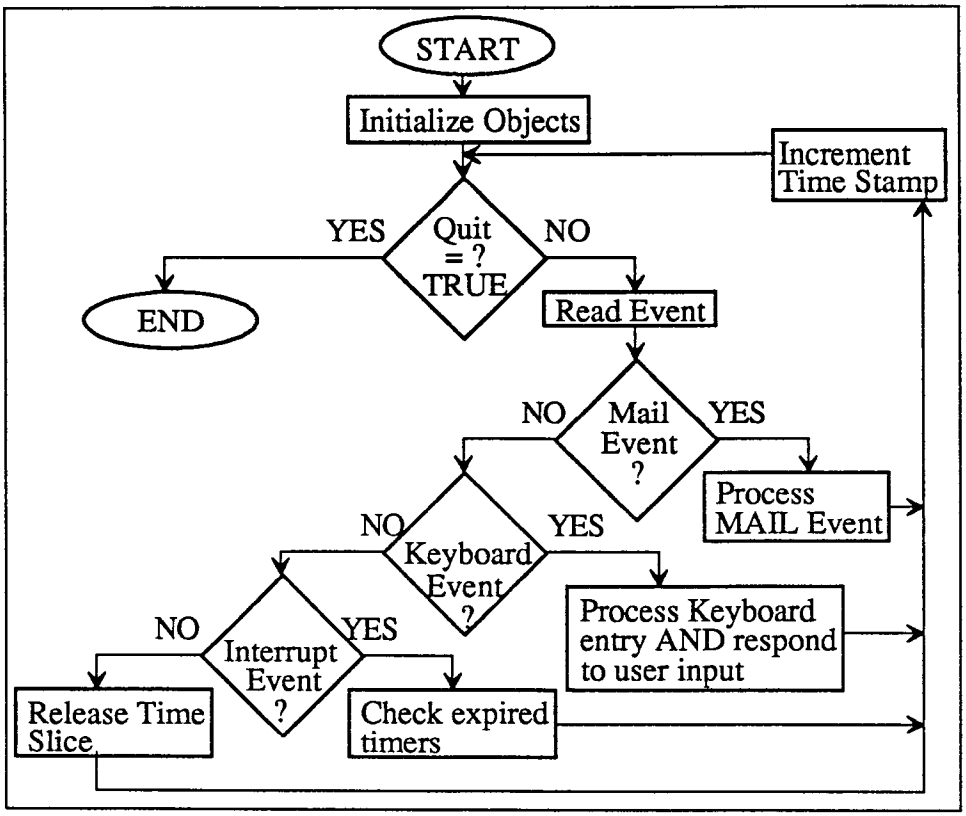

Figure 3.4: BOARD HANDLER's main loop.

3.3 Agent Modules

There are three agent code programs: agent01.c, agent11.c, and agent21.c. When an agent starts its execution, first it waits for a message from the MANAGER. This message tells it how many copies of the same type are already running in the system. Then it declares itself to the handler via writing its name, profession, and mailbox key in the BOARD HANDLER's mailbox. Afterwards, it waits for two events to occur:

- Mail event, and
- Timer event.

When a mail event occurs, it calls the *ProcessMailEvent* function. In this function the following two steps are done: reading and buffering incoming messages, and then processing them according to their context. Whereas in case of a timer event it calls *ProcessTimerEvent* function. This function is executed periodically to look after other timers that exist in the internal data structures.

3.4 Messages and Error Control

Modules across the system communicate using the message passing mechanisms provided by the API services. The sender module creates the message for sending, then, using the mailbox handler of the receiver, it invokes the following function call:

mal_write(MailKey, MessageString, LengthOfString);

or

mal_addto(MailKey, MessageString, LengthOfString, Status);

DESQView queues messages arriving to a mailbox, without reordering, waiting for the receiver to read them. The receiver module can test whether its mailbox is empty or not by the following function call:

NoOfMessages = mal_sizeof(MailKey);

The receiver may use the following function to read from its mailbox:

Status = mal_read(MailKey, &MessageString, &LengthOfString);

Each incoming message consists of the following components:

1. Message string: string of consecutive bytes. The pointer passed to the function returns with the starting address of this string.
2. Status: a one byte integer associated with each message. One can send a NULL message with a status integer.
3. Mailbox handler of the sender: this information is available, directly after reading the message, through the function call:

MailHandler = mal_addr(MailKey);

In the current system the message is either the contents of the string or the status integer, but not both. Later both of them could be used to establish a prioritized message passing mechanism. Unlike the first version of the system, there is no need to lock the communications channel.

3.4.1 Types of Messages While Figure 3.5 shows a schematic diagram for the kinds of messages the system modules exchange, Figure 3.6 depicts message formats. BOARD system messages can be classified into five classes. Administrative messages are used to control the appearance of the system windows on the screen. They are used also to do system-house keeping. The declaration messages are the type of messages that declare the existence of a certain module to another module. The third category is the acknowledgement messages group. These messages are used to implement an exchange protocol between the two parties of the communication process (refer to the next section). The last two groups are command messages and result messages. Command messages are requests (for the receiver) to perform a certain action or computational task. The receiver of a command message usually replies with a result message indicating the outcome of the performed action (if any). While the first three categories are problem independent, the last two are problem specific.

3.4.2 Error Control To increase the reliability of communication channels, a modified ARQ [10] protocol is used. Packages of messages are stamped with a

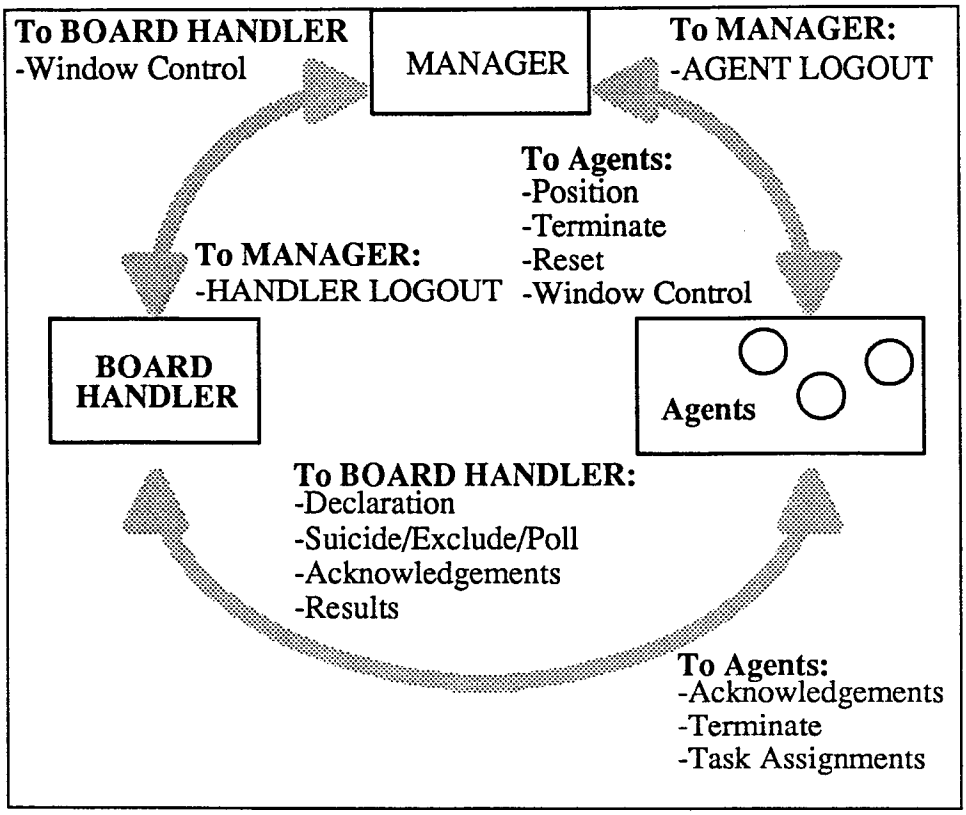

Figure 3.5: Messages exchanged among modules.

time stamp, and a sender does not suspend itself waiting for an acknowledgement. It proceeds but advances the time frame indicator to mark a new time frame. The module keeps a copy of the messages it sent in case it is required to resend them.

Another control mechanism is the time-out timers. A sending module expects the a response from the receiver within a predetermined time window. If the time-out timer expires the sender re-sends its messages. This is repeated until the messages are acknowledged or the maximum time for keeping the message copies has past.

4 PERFORMANCE

One goal of optimizing the system is to reduce the penalties for using message passing and decomposing the problem. Obviously, these penalties cannot be nil. Message passing introduces channels of communications that have to be maintained. Furthermore, queuing and parsing messages are extra tasks to be performed by the receiver. Coordinating among the on-going activities and putting final pieces of the solution together is not a trivial task.

It is shown in [9] that the BOARD system can be modeled as an N-pipeline system as shown in Figure 4.1. The turn-around time can be estimated using the following equation:

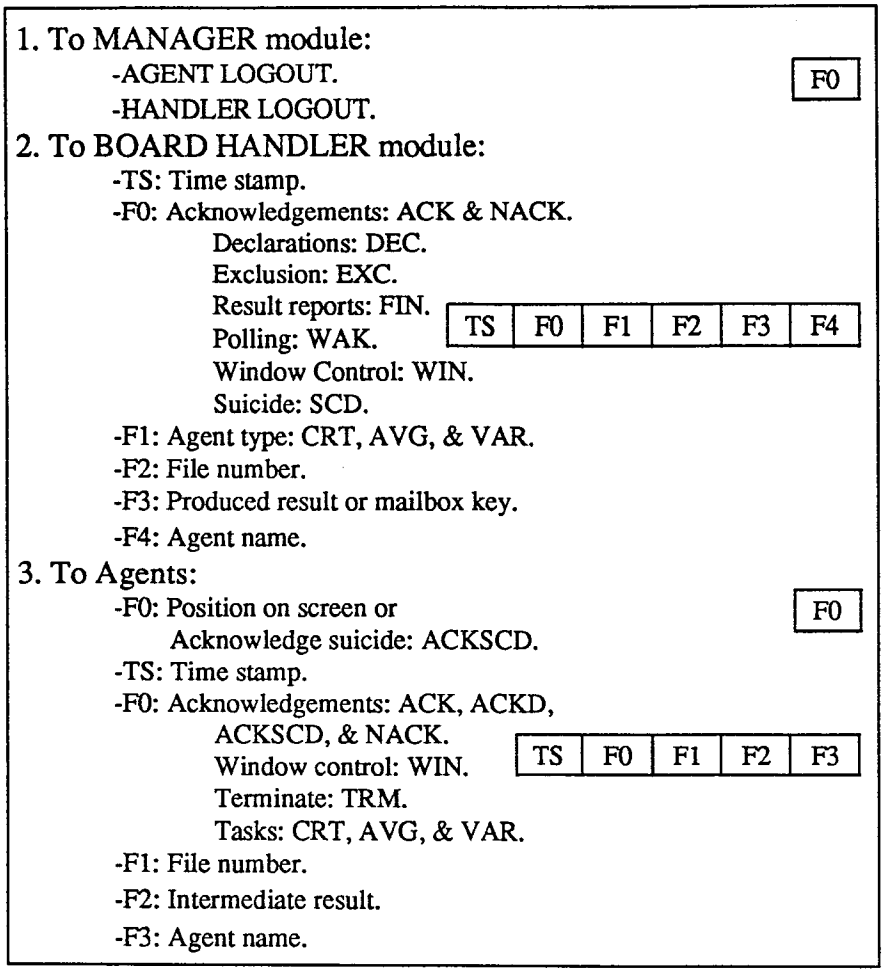

Figure 3.6: Message formats.

$$T(N,\tau) = T^* + M(\tau,T^*)T_{dv}[2+3N] + T_{board}$$

Where N is number of copies of each agent functioning in the system. τ is the time slice (in ticks) given to each modules. T^* is the clean time without DESQView or BOARD system overhead. $M(\tau,T^*)$ is a constant of value 1 or 2 for the implemented problem. T_{dv} is context switch time for DESQView. T_{board}

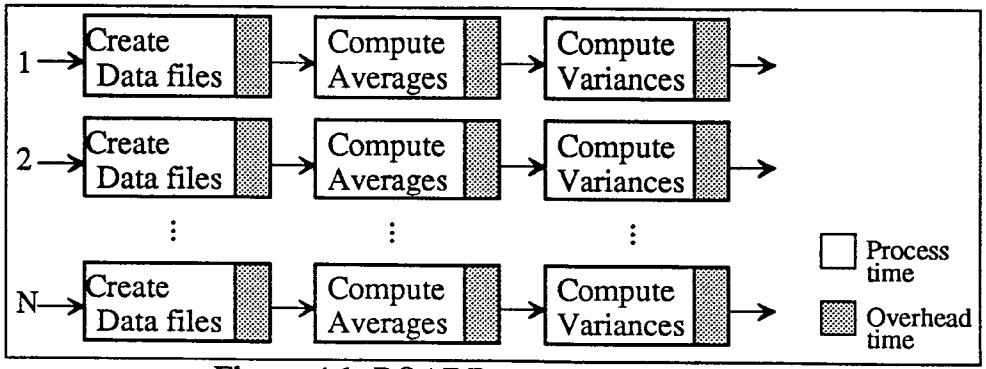

Figure 4.1: BOARD system, N-pipelines.

time spent in the BOARD HANDLER module and it is almost independent of the agents' primary tasks.

For the current problem and implementation, 24 percentage extra time was required to perform the problem compared to a straightforward code. Four percent of this degradation was due to context switching in the DESQView environment while the rest is due to the introduction of the BOARD HANDLER stage. Message passing using DOS files was approximately four times slower than that using API mailboxes.

5 DISCUSSION AND CONCLUSIONS

System degradation is a function of both the environment and problem selected. In the presented problem (dealing with data files) a great percentage of time was spent doing message passing, because the actual job to be carried out was trivial and does not take significant time. It can be argued that the implemented prototype deals with a semi-extreme case where the intensity of message passing is very high. In a real situation, an agent would spend a significant amount of its time doing its primary task and even in that case the overhead is expected to remain almost constant since it is dominated by the degradation which is due to the BOARD HANDLER. Hence we conclude that, degradation in performance is not a central issue. In situations where heavy traffic of messages start to rush into upper levels, a message overload model would be able to filter only those that represent major or critical events.

It is true that there is an overhead due to the process of tailoring the problem to conform with the described configuration, but there are advantages as well. Tasks in the suggested configuration can proceed in parallel and more control can be exercised over primitive tasks, if necessary. The fact that the overall problem solving is pursued in parallel threads enables the system to backtrack from a certain course of investigation in case a more critical event appears.

DESQView is a convenient test-bed for implementing different system configurations and for investigating ideas. In spite of its simplicity, it provides most of the required functionality (concerning process control and inter-process communication) provided by other operating systems but with less complexity. Furthermore, it keeps the door open for the whole variety of DOS tools since it is DOS compatible.

6 FUTURE DIRECTION

The intention is to continue this exploration and investigate any other possible bottlenecks of the model. Success in this task would lead to a better implementation of the overall problem solution. Another goal is to implement the Secondary Side Chemistry problem (at Pt. Lepreau) as an illustration of

system operation. Exploring other platforms in a multi-processor environment is also possible.

7 ACKNOWLEDGEMENTS

This work has been made possible through funding from the National Sciences and Engineering Research Council of Canada, with further staff support from the New Brunswick Electrical Power Commission. Special mention and thanks go to Bryan Patterson, NB Power, for suggesting the application and for his technical support.

REFERENCES

1. Private communication(March 21, 1991), Mr. James, M. Licensing Assessment Officer, Radioisotope and Transportation Division, Atomic Energy Control Board, Ottawa, Canada.
2. Olmstead, R.A., Pauksens, J., and Goodyn, J. *"New Approaches to Alarm Annunciation for Candu Power Plants"*, pp.3-15:3-21, Canadian Nuclear Society Proc. 10th Annual Conference, 1989.
3. Naito, N., Sakuma, A., Shigeno, K., and Mori, N. "*A Real-time Expert System for Nuclear Power Plant Failure Diagnosis and Operational Guide*", Nucl.Tech., 79, pp.284-296, 1987.
4. Suzuki, J., Sueda, N., Gotoh, Y., and Kamiya, K. "*Plant Control Expert System Coping with Unforeseen Events — Model-based Reasoning using Fuzzy Qualitative Reasoning*", pp.431-439, Proc. Third International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (Sponsored by ACM), 1990.
5. Broeders, H., Bruijn, P., and Verbruggen, H. "*Real-time Direct Expert Control using Progressive Reasoning*", Eng. App. of AI, 2(2), pp.109-119, 1989.
6. Coyle, F. "*Expert Systems — Ready for Real-time*", IEEE Expert, 5(4), p.12, 1990.
7. Slack, M.G. "*Situationally Driven Local Navigation for Mobile Robots*", JPL Publication #90-17, 193p, April, 1990.
8. Ionescu, D., and Trif, I. "*A Hierarchical Expert System for Computer Process Control*", Eng. App. of AI, 1(4), pp.286-302, 1988.
9. Mahmoud, A.S. "*Distributed Problem Solving in an Engineering Domain*", Masters thesis (in preparation), Department of Engineering Physics, McMaster University, 1992.
10. Stallings, W. "*Data and Computer Communications*", Macmillan Publishing Company, 1988.