Estimating Resource Requirements of Real-Time Actor Systems Through Simulation

by

Sanjay Kohli

project submitted to the faculty of Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Masters of Science

in

Computer Science

APPROVED:

Dr. Dennis G. Kafura, Chairman

Dr. Richard E. Nance

Dr. James D. Arthur

C.2

Abstract

The major objective of this project is to estimate the resource requirements of a real-time program developed in the actor-based system ACT++. This report describes the design and implementation of *instrumentation primitives.* Using these primitives, a real-time system can be simulated and an estimate of processor and memory requirements can be obtained.

Acknowledgments

I wish to express deep gratitude to Dr. Dennis Kafura, without whose patient guidance and invaluable technical advice this work would not have been possible. I would like to thank Dr. James Arthur and Dr. Richard Nance for serving on my committee. I would also like to sincerely thank Keung Hae-Lee for guiding me at critical times. Finally, I would like to thank the other members of my group, particularly Greg Lavender, for encouragement throughout this work.

Table Of Contents

List of Illustrations

1. Introduction

The use of computers for controlling and monitoring industrial and military applications has increased over the years. Many of the applications in these areas require real-time response from computers. With their increasing processing power, the computers are shared among a number of time-critical control and monitor functions. It is useful to think of these functions as tasks that have their own thread of execution. The tasks are basic entities of a real-time system: they are assigned resources and are responsible for performing specific functions within a given deadline. The scheduling of tasks so that they meet their deadlines is an important issue.

Tasks in our system consist of concurrently executing actors which interact via message passing to carry out the work of the real-time systems [Kafura 88]. As mentioned above, the tasks in real-time systems have deadlines. Since a task in our system is a sequence of steps performed by various actors, each actor is given a deadline by which it should complete its computation. To simplify our system, all actors in a task have the same deadline, which is the same as that of the task.

Subsections 1 and 2 of this chapter give an overview of real-time systems and discuss the issues involved in scheduling real-time tasks. Subsection 3 discusses the design of instrumentation primitives, and subsection 4 describes the goals of this project.

1.1. Overview of Real-Time Systems

When characterized by their deadlines, tasks are usually divided into two categories, namely, *soft* real-time tasks and *hard* real-time tasks [Stankovic 88]. A hard real-time task has to be performed not only

correctly, but also in a timely fashion to guarantee the validity of certain assertions about the timing of events. Typically an assertion involves the maximum response time to an external stimulus, or the periodic update rate of the process variables under control. A task failing to meet its timing requirements can have serious consequences for certain types of systems, such as space shuttle avionics systems, flight control systems, missile launchers, or other defense systems. On the other hand, the consequences of missing a task's deadline in a soft real-time system are not so drastic. One of the major goals of soft real-time systems is to execute the tasks as quickly as possible, thus improving the responsiveness of the system.

Most multitasking real-time systems have a mixture of hard real-time tasks and soft real-time tasks [Hatley 87]. The hard real-time tasks must respond to certain input by a stated deadline, while soft real-time tasks have no hard deadline. Even among the tasks with hard deadlines, there are cases in which a missed deadline can be tolerated. An example of such a task would be one which periodically takes the temperature reading in a shuttle system. In this case, even the failure of the task to complete its work by a certain deadline could be tolerated because penalties for missing the deadline are not very high (drastic). Thus, real-time tasks should be characterized by both their deadlines and the penalties for missing deadlines (but usually the penalties are not considered in the literature [Liu 73, Mok 84]).

The second major characteristic of a real-time task is its arrival characteristic. In the context of scheduling, the tasks could be divided into two types: *periodic* and *aperiodic*. Periodic tasks arrive at regular intervals, whereas aperiodic tasks are triggered by asynchronous events outside the system. In both cases, however, the scheduling goal is the same: to schedule the task so that it is completed before its deadline.

1.2. Scheduling Real-Time Tasks

To schedule real-time tasks, the execution times and the period (for periodic tasks), or the arrival times and the deadlines (for aperiodic tasks) must be known. The design of a real-time system is a two-step process. The first step consists of estimating the resource requirements of individual tasks that comprise the system. The second step determines the feasibility of scheduling the set of tasks based on the resources that the system supports.

The feasibility of scheduling a periodic task set can be found by calculating the load on the system, using the formula Σ_i (C_i / P_i) where C_i and P_i are the computation time and period, respectively, of task i [Liu 73]. A task set is *schedulable* if all the deadlines of all the tasks can be met indefinitely. If the sum (the load of the system) is less than 1 (equivalently the load is less than 100%), the task set is schedulable. If the sum exceeds 1, there is no guarantee that all the tasks will be completed before their deadlines. If tasks cannot be guaranteed to meet their deadlines, changes must be made in the design of the individual task or the system to comply with applications timing requirements.

1.2.1. Analytical versus Simulation Technique

This report is concerned with calculating the computation time and memory requirement of a task. To calculate the computation time C_i, we focus on the behavior of a single task in the system. We can either use an analytical technique or a simulation technique to determine the computation time. The next section briefly explains an example of an analytical technique described by Woodbury [Woodbury 86]. We will also explain why this approach, in particular, and an analytical technique in general, is not suitable for our purpose. A later section describes the simulation technique used in this work.

1.2.1.1. Analytical Method

Woodbury describes a method to model the structure of real-time tasks, whose direct analysis produces a probability distribution function (PDF) and a probability density function (pdf) of the execution time of a task. A task is considered to consist of Q *steps*, where a step may be a machine cycle, an instruction, or a procedure. Depending on its function, a step can be classified as an input, an output, or a computation step. The structure of a task is defined using an *Execution Structure Diagram* (ESD). The ESD represents the computational portion of a task. In the ESD, random variables are associated with input - I, output - O, computation - C, or recursive - R steps. I and O characterize the time request for input/output operations and C the internal execution time of non-I/O steps. The component R is a recursive element that can be replaced by any combination of the other three components. The computation step C further consists of sequential step S_i and branching step B_i. The Branch set utilizes AND/OR operations on sequential or recursive components. One example of ESD is shown in Figure

1(a). From this we derive the *Computation Path* of a task which consists of sequential and branch instructions as shown in Figure 1(b). We can then associate an execution time t_{Si} with the i-th sequential set of instructions and t_{Bi} for branch set Bi. The term t_{Bi} is calculated by first finding the product of the probability for the branch and the execution time for that branch, and then summing this product over all the branches in Bi. From this we can derive the probability distribution function, F_C (t), for computation path,

$$F_C(t) = P[C < t],$$

$$F_C(t) = P[\Sigma_i (t_{Si} + t_{Bi}) < t]$$

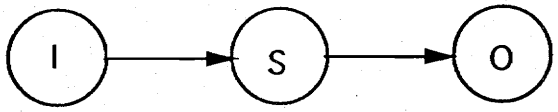
where F_C (t) is used further to calculate the PDF for a task.

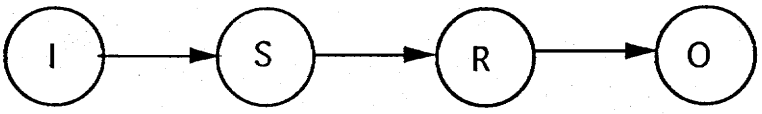
This scheme is based on a model where a system can be represented by static graph structures and each component can be precisely identified as an input, output, sequential, or branch instruction. There are two reasons why this technique is not suitable for an actor-based system:

- The non-determinism of an actor-based computation. This non-determinism results because the order of arrival of messages is indeterminate. In particular, when messages are sent from one actor to another, the arrival ordering at target may not correspond to the order in which they were sent [Agha 86]. It means that in actor model a computation may follow an indeterminate path.

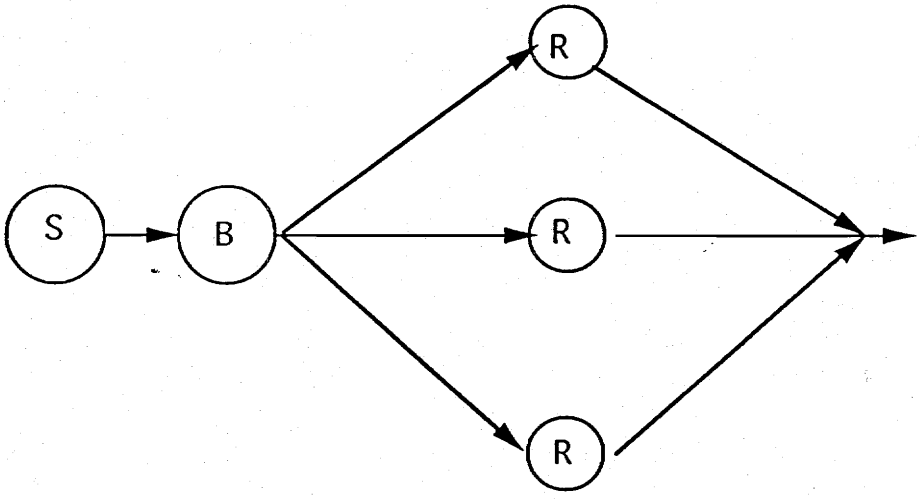
- Woodbury's scheme is control-driven (a task corresponds to a control flow path), whereas an actor-based system is message-driven (a task is a pattern of messages among concurrent actors).

Thus analytic techniques are currently suitable only for sequential systems and further work needs to be done to make them applicable for concurrent message-driven systems. Because of this, we use the simulation technique, which is much simpler to implement.





1(a)



1(b)

Figure 1. Execution Structure Diagrams

1.2.1.2. Simulation Technique

In this scheme, a real-time system designer specifies the expected computation time at each step in a task and the probabilities of performing these steps. This task is then "run" several times with different inputs to obtain statistics about average execution time and worst-case execution time. Next, the user builds a simulated system consisting of a number of the simulated tasks. The worst-case execution time of the tasks, gathered in the first step, is used to calculate the load on the system. The simulated system gives the designer useful information related to scheduling feasibility. The simulation approach has the following advantages over the analytical approach:

- The dynamic nature of computation in actor systems is accounted for. A computation in an actor system is achieved by actors communicating via message passed in response to the communication sent to the system. As computation proceeds, an actor system evolves to include new actors that are created as a result of processing actors already in the system. The resource requirements for this system are easily assessed in the simulated environment.

- Early performance analysis is obtained. This analysis is incremental and progressive and can be carried out at various levels of refinement during the design phase. The simulated runs can give the user a better understanding of the run-time behavior of the system. This information can be used to improve the design of the system, redesign a particular task, or use a different approach.

- A basis is provided for debugging and visualization of a real-time program. This further enhances a designer's capability to understand the real-time system being developed. These extensions will be discussed in Chapter 6.

1.3. Instrumentation

The execution characteristics of a task are specified through a collection of instrumentation primitives that are implemented as an extension of the ACT++ class hierarchy. ACT++ is an actor-based system developed at Virginia Tech [Kafura 90]. ACT++ is based on C++ and has predefined system classes that

support the primitive operations of actor systems. The details of ACT++ system classes are given in Chapter 3.

The ACT++ system is extended to provide instrumentation primitives by adding properties to each of the ACT++ classes. In the present case, the properties that are added allow statistics about execution time and memory usage to be gathered. This is done by creating an *instrumented* class for each ACT++ class. The objects created as instances of instrumented classes are called *instrumented objects.* The instrumented classes are subclasses of ACT++ classes. In addition, there are two other classes provided to instrument the C++ conditional and sequential constraints. When any operation is performed on an instrumented object, it updates a simulated clock to reflect the time taken for the operation. Similarly, when an instrumented object is created the memory usage statistics are updated to reflect the memory requirement for that object.

The scripts written with the instrumentation primitives are called *instrumented scripts.* The user creates an instrumented script for each actor in a task. These instrumented scripts are executable code and can be run to simulate the task's execution. The execution of these scripts gives an estimate for execution time and memory demand for a task. An *instrumented system* is built from instrumented scripts. This process is explained in section 4.2.

The instrumented classes, created by C++ inheritance, allow the user to create an instrumented system that has the same structure as the completed system. Therefore, each instrumented class can be replaced by an ACT++ class or C++ code at successive phases in development. Thus an instrumented system is not a distinct and separate entity from the actual system, but becomes the final system by successive development.

1.4. Goals

One of the goals of this project is to provide the instrumentation primitives with which a real-time system developer can calculate the time taken by different computational steps in a task. It also allows the developer to obtain information about memory requirements of the system. The developer writes

instrumented scripts using these primitives. These scripts can then be executed to give the estimates of the execution times and memory requirements of a task.

This report also describes the changes that were made to ACT++ to make it suitable for developing real-time systems. It describes the new primitives that were added to ACT++ to allow creation of the real-time tasks and changes that were made to ACT++ to incorporate a priority-driven scheduler.

1.5. Overview of the Report

Chapter 2 of this report describes various characteristics of real-time tasks and discusses different scheduling strategies described in the literature. Chapter 3 gives a short description of the actor model of computation as proposed by Hewitt. This chapter also describes an actor system, ACT++. Chapter 4 describes the design of instrumentation primitives and a scheduler. Chapter 5 gives an example of a program written using the instrumentation primitives. Chapter 6 summarizes the results and suggests directions for future work.

2. Scheduling

The function of a scheduling algorithm is to determine, for a given set of tasks, whether a schedule (the sequence and time periods for executing the tasks) exists such that the timing, precedence, and resource constraints of the tasks are satisfied, and to calculate such a schedule if one exists.

Task scheduling in hard real-time systems can be *static* or *dynamic*. In a static scheduling scheme the schedule for the tasks is determined off-line. This scheduling scheme requires complete prior knowledge of the tasks' characteristics. Tasks in a dynamic scheduling scheme can arrive arbitrarily and are scheduled on-line and progressively. Although static approaches have low run-time costs, they are inflexible and cannot adapt to a changing environment. When new tasks are added to a static system, the schedule for the entire system must be recalculated. In contrast, the dynamic approach is more flexible but involves higher run-time costs, because the scheduler has to adapt its schedule every time a new task is added to the system.

A static scheduling algorithm is said to be *optimal* if, for any set of tasks, it always produces a schedule which satisfies the constraints of the tasks whenever any other algorithm can do so*. A dynamic scheduling algorithm is said to be *optimal* if it always produces a feasible schedule whenever a static scheduling algorithm with complete prior knowledge of all the tasks can do so [Stankovic 88].

---

* An optimal algorithm need not give us the best solution but it always produces a feasible schedule. The definition does not refer to optimizing any variable but rather satisfying certain constraints about time.

2.1. Task Characteristics

Schedulers in real-time systems are typically based on the following task parameters:

- the arrival time, the time at which a task either arrives in system or is invoked in the system,
- the ready time, the earliest time at which a task can begin execution,
- the worst-case computation time of the task, and
- the deadline, the time by which a task must finish its execution.

A scheduling strategy which varies all of these parameters simultaneously is very complicated. Thus, most schedulers make certain assumptions regarding the variation of these parameters within themselves or with respect to other parameters. For example, the earliest-deadline-first algorithm assumes that the deadline of a task is the period between successive arrival times of that task. Although several combinations are possible, the following task categories are broadly used in different scheduling strategies.

2.1.1. Periodic versus Aperiodic Tasks

An aperiodic task is one whose arrival is not known in advance. As with other real-time tasks, an aperiodic task has a deadline by which it must finish its computation. Periodic tasks arrive in a fixed, known pattern (i.e., the interarrival period is constant). The deadlines of periodic tasks are always less than or equal to the periods, so a given instance of a periodic task must finish its execution before its next (instance) arrival. The arrival time and the deadline of instances of a periodic task with period P are specified as follows:

$$A(i+1) >= D(i)$$

$$D(i+1) <= A(i+1) + P$$

where A (i) and D (i) are the arrival time and the deadline of the i-th instance of the periodic task, respectively.

Examples of periodic tasks include activities that need to be done regularly, like temperature sensing in a control system. An example of an aperiodic task is a task that takes some action as a result of a sudden change in systems characteristics.

Most schedulers consider the task system to consist entirely of periodic tasks [Liu 73]. If the system has both kinds of tasks, schedulers normally consider only periodic tasks and leave certain *slots* for aperiodic tasks, in which they are scheduled to run. However, if there are no aperiodic tasks to be scheduled during a certain interval of time, these slots go empty and the processor runs at reduced utilization.

2.1.2. Preemptable versus Nonpreemptable

A task is preemptable if its execution can be interrupted by other tasks and resumed afterward. A task is nonpreemptable if it must run to completion once it begins. Whether a task is preemptable or not is mainly determined by the nature of the application environment.

2.1.3. Precedence-Constrained versus Independent

A task t_i is said to *precede* task t_j if t_i must finish before t_j begins. Independent tasks have no precedence relation between them. Such tasks can run independently of each other, one after another (in any order), or simultaneously.

2.2. Scheduling Algorithms

The following is a brief survey of different scheduling algorithms. The classification is based on the task's characteristics mentioned above. Figure 2 summarizes these algorithms.

2.2.1. Static Scheduling Algorithms

We will discuss scheduling of mutually independent tasks and tasks with certain kinds of precedence relations. The two different types of systems -- uniprocessor and multiprocessor -- are considered in the following discussion.

2.2.1.1. Scheduling Mutually Independent Tasks

We distinguish between schedulers where preemption is allowed and schedulers where tasks are nonpreemptive.

If *preemption* is allowed for a set of tasks with arbitrary ready times and deadlines, then the simplest scheduling scheme is the *earliest-deadline-first* scheme. In this scheme, the tasks with earlier deadlines

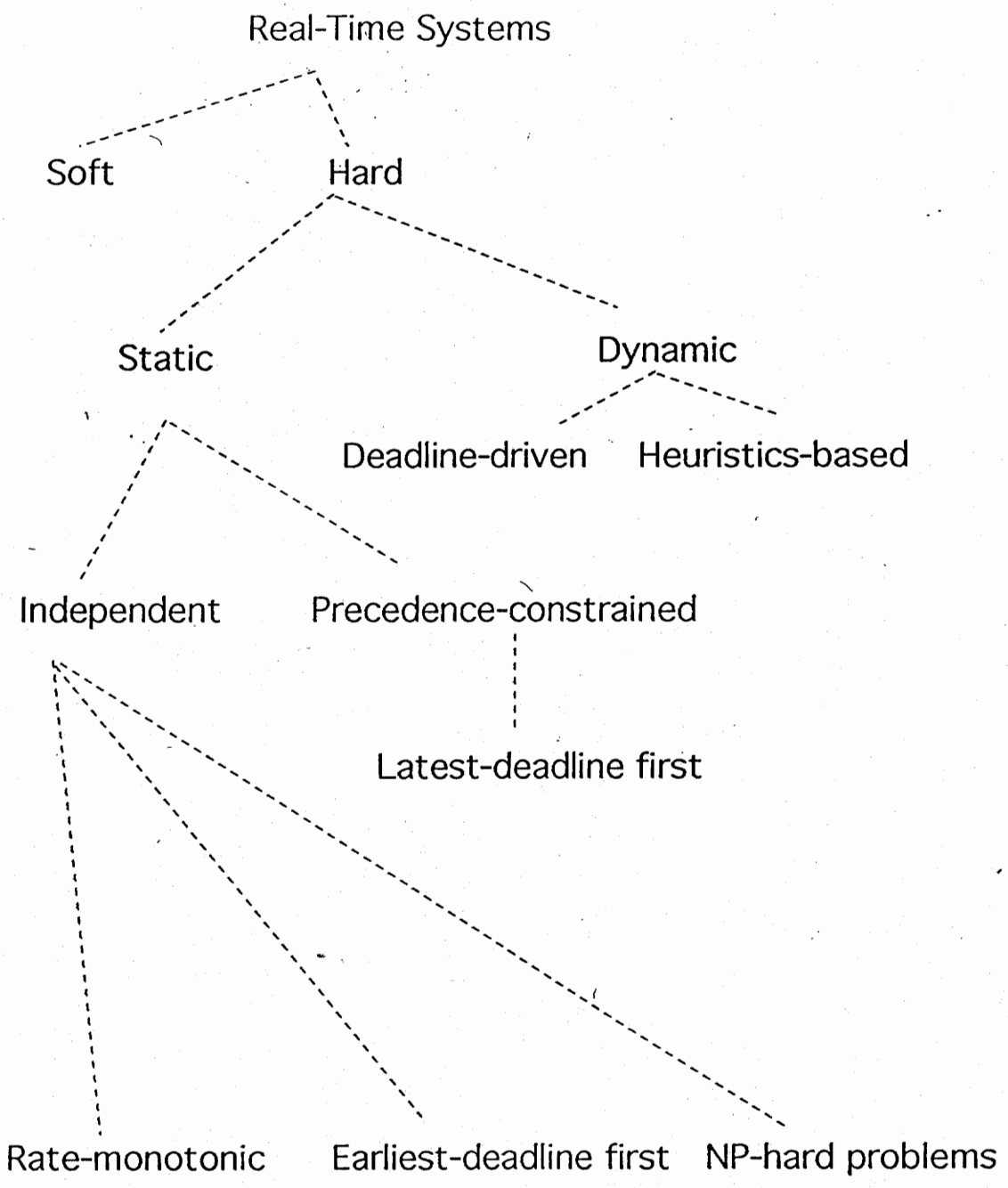


Figure 2. A Taxonomy of Real-time Scheduling Algorithms

and ready times are chosen to run before tasks with later deadlines and later ready times. This is an $O(n^2)$ algorithm, where n is the number of tasks to be scheduled.

Liu and Layland [Liu 73] developed a scheduling scheme for periodic tasks on a single processor. This scheme is called the *rate-monotonic priority* scheme; it assigns higher priorities to tasks with shorter periods. Thus a task, τ_1, with a shorter period than another task, say τ_2, is scheduled to run earlier than τ_2. If another task τ_3 arrives when τ_1 is running and if the new task has a shorter period, then τ_1 is preempted and τ_3 is scheduled to run. It can be shown that this scheduling scheme is optimal by considering two tasks, τ_1 and τ_2, with computation times C_1 and C_2 and periods P_1 and P_2, respectively. Also assume that $P_1 < P_2$, then if higher priority is assigned to τ_1, the following inequality needs to be satisfied,

$$\lfloor P_2 / P_1 \rfloor C_1 + C_2 <= P_2 \qquad (1)$$

This inequality means that τ_1 will be executed $\lfloor P_2 / P_1 \rfloor$ times ($\lfloor P_2 / P_1 \rfloor$ is the arrival rate of τ_1 in period P_2) while τ_2 should at least be executed once in period P_2. However, if we assign higher priority to τ_2, then we have the following inequality,

$$C_1 + C_2 <= P_1 \qquad (2)$$

Both the tasks should finish their execution before another instance of τ_1 arrives. An optimal solution may exist even if (2) does not hold. However, by using the second assignment, τ_1 may miss its deadline, while an assignment where τ_1 is given higher priority always gives us the solution. This example can be extended to a system with n tasks.

The rate-monotonic algorithm is perhaps the most widely known real-time scheduling algorithm, but it is still not a widely used algorithm because of some practical problems in its application. Sha et. al. [Sha 86] point to the problem of scheduling instability. The algorithm works well if the stochastic execution

time matches well with the average task execution time, but if the tasks take longer to execute than estimated, the system load increases and certain tasks will then miss their deadlines and make the system (and thus the schedule) unstable. The way around this problem is to use worst-case analysis rather than average-case analysis. Since worst-case execution time is typically much larger than average-case execution time, the processor utilization is very low when worst-case analysis is used. So, they suggest using average-case analysis but increasing the priority of important tasks by subdividing the tasks into several parts. They distinguish between the tasks whose deadlines must be met and the tasks whose deadlines can be missed in transient overload conditions. A task which has a hard deadline can be subdivided into different parts so that each part could be scheduled separately at higher priority than a task with a soft deadline.

In *multiprocessor* systems, a set of tasks can be partitioned among the individual processors and can use either the rate-monotonic or the earliest-deadline-first approach to schedule the tasks on each processor. The multiprocessor scheduling problem in this case is known to be an NP-hard problem. Thus, heuristics are used to partition the task set. The best-fit, first-fit, and next-fit heuristics are some of the common approaches to achieve the partition [Davari 86].

On uniprocessor systems, the earliest-deadline-first algorithm is optimal for *nonpreemptive* tasks if the tasks have the same ready time. However, most of the scheduling problems are NP-hard for nonpreemptive tasks. It can also be shown that a uniprocessor nonpreemptive scheduling problem is NP-hard if the tasks have arbitrary ready times. On multiprocessor systems, the problem is NP-hard even if the tasks have the same ready times [Ullman 75].

2.2.1.2. Scheduling Tasks with Precedence Constraints

On a uniprocessor system, it can be shown that scheduling nonpreemptable tasks with deadlines and arbitrary precedence constraints can be solved using the *latest-deadline-first* algorithm in $O(n^2)$ time. Tasks are scheduled from last to first, one at a time. At each step in the algorithm the task with the latest deadline is chosen among those whose successors have been scheduled.

The earliest-deadline-first algorithm can also be used if the ready times and deadlines of a set of tasks with precedence constraints are modified such that the modified times comply with the precedence constraints of the original tasks. Therefore, in this case, precedence need not be considered explicitly.

2.2.2. Dynamic Scheduling Algorithms

Unlike a static scheduler, a dynamic scheduler has no *a priori* knowledge of the arrival of tasks. Most of the static scheduling algorithms can actually be run on-line, but they do not guarantee optimal scheduling. Mok [Mok 84] calls a run-time scheduler *clairvoyant* if it can predict the future arrival times of the tasks with certainty. A run-time scheduler is optimal if it produces a feasible solution every time a clairvoyant scheduler can do so.

One algorithm that does produce an optimal solution is called the *deadline-driven* scheduling algorithm [Liu 73]. In this algorithm, priorities are assigned based on deadlines of the tasks. A task with an earlier deadline is assigned a higher priority than one with a later deadline. The priority assignments are done whenever a new task arrives in the system. It is easy to see why this algorithm is optimal. Processor utilization can reach 100% with this strategy.

To understand why this algorithm is optimal, we assume two tasks, A_1 and A_2, with execution times C_1 and C_2 and deadlines D_1 and D_2, respectively, with $D_1 < D_2$. If task A_2 is scheduled earlier than A_1 and if $C_2 > D_1$, then A_1 will miss its deadline. But if A_1 is scheduled first, both the tasks will meet their deadlines. Note that if the system load exceeds 100%, then at least one of the tasks misses its deadline. In this case the system will make its scheduling decision based on the penalty for missing a deadline. Assuming that the system load is less then 100%, it can be guaranteed that both tasks always meet their deadlines using this scheduling technique. The deadline-driven scheduling scheme is used in our system. In our case a task may consist of several actors. All the actors in a task have the same deadline; the deadline of the task.

However, the deadline-driven algorithm is non-optimal if tasks are nonpreemptable. Suppose task τ_2 *just* arrives in the system when task τ_1 has already been scheduled to run in the system and the execution time

of τ_1 is greater than the deadline for τ_2. In this case τ_1 misses its deadline. An optimal solution could be obtained if the scheduler were clairvoyant and waited for τ_1 to be scheduled before τ_2.

We can use the same argument to show that this algorithm is non-optimal for a multiprocessor system even if the task set is preemptable. Both the above arguments are due to Mok [Mok 84].

3. Actor Systems

The actor model was developed by Hewitt at MIT [Hewitt 76] and elaborated by Agha [Agha 86]. It originated from the λ-calculus and object-based programming. An actor is a computational agent which accepts a message and then performs some or all of the following actions:

- *sends* a finite number of messages to other actors,
- *becomes* an actor with a new behavior, and
- *creates* new actors.

The two basic elements of the actor model are messages, which are also called tasks, and behaviors (or scripts). The computation in an actor system is achieved by actors communicating through messages. If an actor needs a sub-task performed, it sends the message to an actor capable of performing the sub-task (if that actor exists) or creates the actor and then sends the message to that actor. A message may contain the identity of a *continuation* actor to which the results of a computation should be sent. A newly created actor can execute concurrently with the creator actor. When an actor has performed its computation, it sends the results to the actor indicated in the *request* message.

Message passing in an actor system is asynchronous; the receiver does not have to be ready to accept the communication when the sender sends it. To implement asynchronous message passing, each actor has an associated mailbox. All the messages sent to an actor are queued in its mailbox. An actor system has

no explicit *receive* primitive to retrieve the message from its mailbox. Rather, when an actor is created it is automatically ready to receive the message from its mailbox, and if the mailbox is empty the actor is blocked until some message arrives. The actor system does not guarantee *prompt* delivery of the message, it only guarantees *eventual* delivery of the message.

A behavior can receive only one message. After receiving the message, the actor declares its *replacement* behavior, which receives the next message in the mailbox. The replacement behavior can execute concurrently with the original behavior. Concurrently executing behavior is one way in which the actor system achieves its concurrency.

The behavior of an actor is defined by its local data and script (or program). Simple actors (called primitive actors) have the same behavior throughout their lifetimes. Each time they are invoked with the same input data, they produce the same result. Thus, an actor 5, when called to add 7 to itself always returns 12. More complex actors are like processes; they have state information along with local data and a script. The state information is contained in the set of *acquaintances*. The change of state is represented by change of acquaintances. When an actor declares its replacement behavior, it passes the appropriate set of acquaintances to its replacement behavior. These kinds of actors change their behavior over time.

3.1. Actors in ACT++

ACT++ is a concurrent object-oriented class hierarchy developed at Virginia Tech [Kafura 90]. The class hierarchy creates a "language" which is a hybrid of the actor kernel language and an object-oriented language, C++. It provides the class structure of C++ and supports the primitive operations of the actor system: send, receive, create, and become. Its main components are described in the following subsections.

3.1.1. Objects

Objects can be *passive* or *active*. Active objects are actors. They have their own thread of execution. All objects that are not actors are passive objects. Passive objects are invoked by a function call in the code of active objects and are not shared among active objects, because concurrent invocation of their methods may cause the state of the passive objects to become inconsistent.

3.1.2. Actors

There is a predefined class ACTOR in ACT++. The system automatically creates an active object if the object is created as an instance of ACTOR class or one of its subclasses. On the other hand, a passive object is created if the instance is not directly or indirectly a subclass of ACTOR. A new actor is created using a *New* operation.

3.1.3. Messages

To support message passing, ACT++ has a predefined class called mailbox. This class has two subclasses, Mbox and Cbox, which provide for two different kinds of messages: *request messages* and *reply messages*. An actor sends a request message to another actor if the sender wants some work to be performed by the receiver. The message contains the data and the method name to be executed by the receiving actor. A message can be sent to another actor using the *send* operation. The send operation needs the identification of the Mbox of the receiver and the message to be sent. Each behavior of an actor reads the message from its mailbox using a *receive* operation. An actor does not have to explicitly use the receive operator. To receive its first message, the actor is activated when a message arrives for it.

A reply message is used to deliver the results of a method invocation to the sender of the request. The reply-destination, a Cbox name, is a part of the request message. The reply operation delivers a reply message to the reply-destination. To read a message from its Cbox, an actor performs a receive operation

on that Cbox. If there is no reply message in the Cbox, the actor is blocked until the message arrives. An actor can use the *in* primitive to check whether any reply message has arrived in the Cbox.

3.1.4. Worker

The worker is a general purpose scheduler that selects among those actors which are ready to run. It is a FIFO scheduler where tasks are retrieved from the scheduling queue in the order they were added. ACT++ differs from the original actor model of Hewitt in that in ACT++ an actor can suspend its execution by waiting for a message in its Cbox. The worker can then schedule the next actor waiting in the scheduling queue.

3.2. Syntax for ACT++ Primitives

The following subsection describes the syntax of various ACT++ primitives.

3.2.1. Send (<<)

The send method is provided by the class Mbox. Its syntax is specified by the stream output operator of C++, <<. The send primitive has the following parameters:

- the address of the mailbox of the actor to which the message is sent,
- the name of the method that is invoked in the receiving actor when it receives the message,
- the reply-destination to which the reply should be sent, and
- zero or more method parameters separated by <<. For example,

```
ADD1 << &ADD::compute << aCbox << par1 << par2
```

sends a message to the actor whose mailbox is ADD1, invoking a method "compute" of class ADD and passing parameters par1 and par2 to the method. The result of the computation is returned to aCbox.

3.2.2. Receive

The receive operation can be performed on both Mboxes and Cboxes. An actor does not have to explicitly perform a receive operation on its mailbox to retrieve the second and the third parameters specified in a send operation. The second parameter (method name) is used to invoke the appropriate method in the receiving actor, and the reply-destination is used by the system to send the reply message. However, the receiving actor has to explicitly read its method parameters. The syntax for receive operation is specified by the stream input operator, >>. Thus,

```
Msg >> rpar1 >> rpar2
```

receives a message from its mailbox and assigns the values of the parameters in that message to rpar1 and rpar2, respectively.

To read a message from a Cbox, an actor has to give the Cbox name as a parameter. The operator for receiving from a Cbox is ~. Thus,

```
~myCbox >> reply1
```

reads the message from the Cbox and assigns it to reply1.

3.2.3. Reply

The reply primitive is used to send the message to the destination that was specified in the reply-destination field of the request message. Thus,

```
reply (rpar1 + rpar2)
```

will send the result of adding rpar1 and rpar2 to the actor which is specified in the reply-destination field of the request message.

3.2.4. In

The receive operation applied to a Cbox is a blocking operation; if an actor performs a receive on its Cbox when there is no message present, the actor is blocked until some message arrives. To avoid blocking, an actor can check whether a message has arrived in its Cbox by using the in operation. Thus,

```
myCbox.in()
```

returns true if there is a reply message in myCbox.

3.2.5. Become

This primitive is used to specify the replacement behavior of the actor. It has one parameter, which is the replacement behavior of the actor. If the actor does not change its behavior, it can specify itself as its own replacement behavior by using Self as an argument.

```
become (new actor_behavior)     // specifies actor_behavior to be new actor behavior
become (Self)                   // specifies itself to be its replacement behavior
```

The become operation plays two important roles in the actor model. This operation is the mechanism by which an actor changes its state. This operation can also be used as a synchronizing mechanism. For additional details please see [Kafura 90].

4. Instrumentation

This chapter discusses the design of the instrumentation primitives that have been provided to build a simulated real-time actor-based system. Section 1 of this chapter discusses the design of these primitives and the changes made to the class structure of ACT++ to achieve this implementation. Section 2 discusses the new primitives that are used to build a real-time system.

4.1. Design of the Instrumentation Primitives

The instrumentation primitives are provided so that the designer can specify the resource requirements of the system. Currently, these primitives give estimates for computation time and memory requirements only. Instrumentation is provided both for ACT++ primitives (which were described in the previous chapter) and basic imperative operations. A script typically consists of both kinds of instrumented operations. The ACT++ primitives include sending a message to another actor, receiving a message from other actors, creating a new actor, or adopting a new behavior. The designer assigns specific units of time to each of the above actions viz. send, receive, create, or become. In addition, the designer needs to specify the time taken by the basic imperative constructs. The basic imperative constructs are sequential and branching actions. Sequential actions are like assignment statements or those statements which involve mathematical or logical computations. Branching actions in a task can be used in *if* or *while* constructs.

In this section we will discuss the new class definitions created to implement the instrumentation primitives. To reduce the number of changes required in moving from an ACT++ program to the instrumented program and vice versa, most of the ACT++ and corresponding instrumentation primitives have the same syntax.

4.1.1. New Class Definitions

This subsection discusses the new class definitions created to implement the instrumented actor system. To estimate the resource requirements for ACT++ primitives -- send, receive, and in -- we create the subclasses of ACT++ classes Mbox and Cbox. The class IACTOR, which is a subclass of ACT++ class ACTOR, gathers statistics for actor creation. To collect information on resource usage, a Resource_monitor class is created. Figure 3 shows the new classes that have been added to ACT++ systems.

4.1.1.1. Resource_monitor

The Resource_monitor class is defined independently of other classes, i.e., it is not a subclass of any ACT++ class. The Resource_monitor object is declared as a *static* object in the file that contains the class definitions of other objects. In this way, C++ creates only one copy of the Resource_monitor object and each class has access to the Resource_monitor as a global object. The Resource_monitor object monitors the utilization of the following resources:

- Time: The Resource_monitor class contains a private variable -- timer. The timer can only be incremented in value and reset to 0. It is updated using the method *update_timer (val)*, which causes the timer value to be incremented by val.

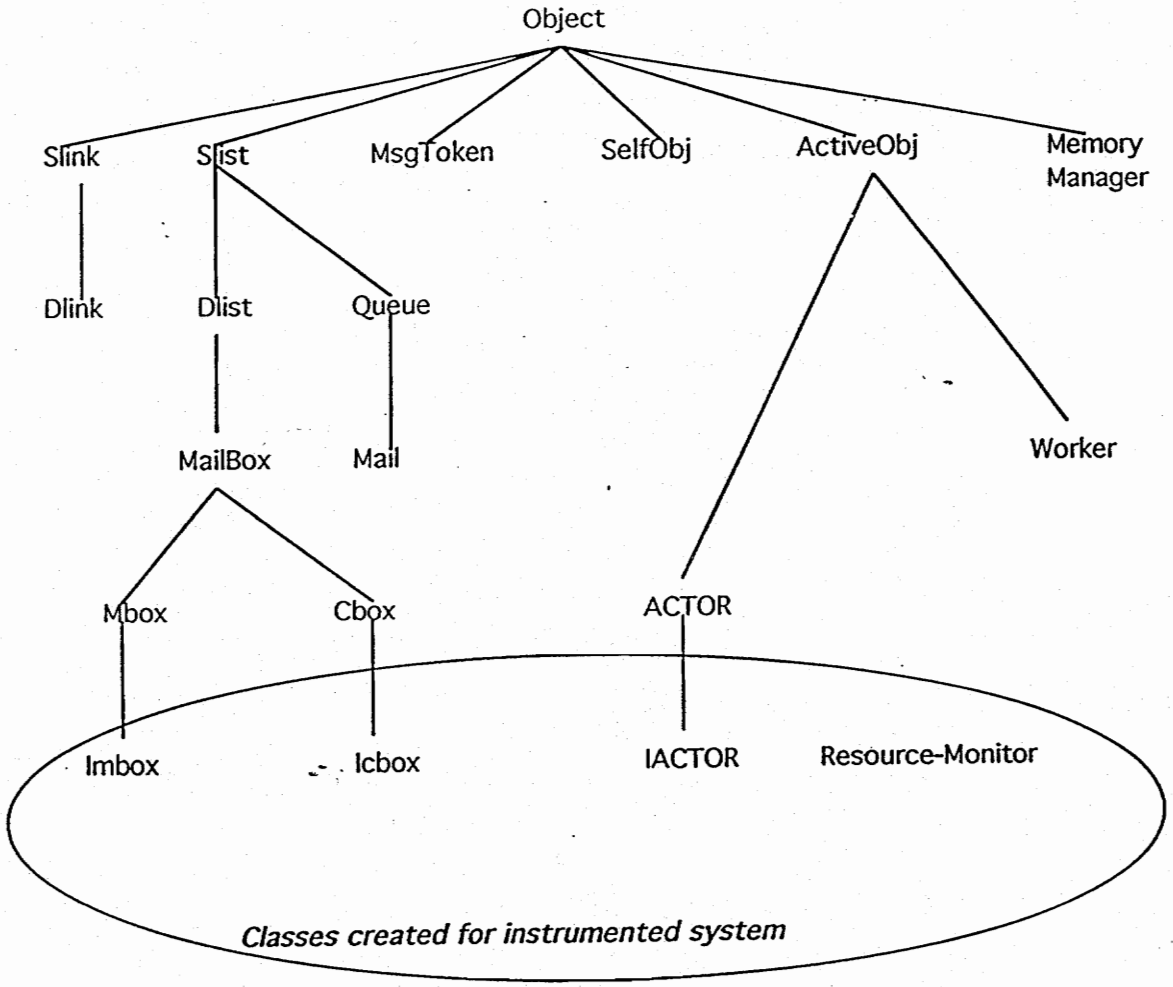


Figure 3. Instrumentation Class Hierarchy

- Memory: A private variable, mem, is updated using the method *update_mem (val)* to reflect the fact that val amount of memory has been requested by the task. Memory may be requested for the creation of an Mbox, a Cbox, and an actor's data area. In addition, an actor also requires a stack area for its execution.

Both variables, timer and mem, can be reset by the method reset().

To determine the resource requirements of an instrumented actor system, the instrumented system is executed several times with different inputs. Before each of these executions, the timer and mem variables are reset using method reset(). At the end of each execution, the statistics about the execution can be collected using the method *collect_statistics()*. The Resource_monitor then updates its variables average_time, average_mem, worst_time, and worst_mem.

4.1.1.2. Imbox

This instrumented Mbox is a subclass of the ACT++ class Mbox. This class provides for methods to send and receive messages. Whenever send or receive primitives are called in the instrumented system, the timer is updated to reflect the time taken by these primitives, after which the send or receive primitive of the Mbox class of ACT++ is invoked. The syntax for the send primitive is the same as in ACT++, a series of stream-like output operations, <<. This operator is extended in the Imbox class to update the timer before calling the corresponding << primitive of Mbox. Similarly, the receive operator updates the timer before calling the >> primitive of Mbox.

4.1.1.3. Icbox

The instrumented Cbox class, Icbox, is a subclass of the ACT++ class Cbox. It provides the methods to reply (reply), read a message (~), or check for the presence of a message in a Cbox (in). The syntax for these primitives is identical to that of ACT++. In the instrumented system, these primitives update the timer before calling their respective methods in the Cbox class.

4.1.1.4. IACTOR

The instrumented actor class, IACTOR, is defined as a subclass of class ACTOR of ACT++. All instrumented actors are instances of IACTOR or a subclass of IACTOR. As earlier, it is still possible to organize a hierarchy of behavior scripts of an arbitrary depth where a subclass reuses definitions contained in its superclasses. The become operation is an overloaded method in the IACTOR class. This operator updates the timer and calls the become operation of ACTOR class of ACT++. The syntax for the become operation is the same as in ACT++.

4.1.2. New Primitives

As discussed in Chapter 1, the actors in our system have deadlines. To create an actor with a deadline, a new primitive, *Inew*, has been introduced. The primitive -- *block* allows the designer to specify the computation time for imperative instructions. The implementation and usage of these primitives is explained in this section.

4.1.2.1. Inew

To create an instrumented actor, a new primitive Inew has been provided. This primitive takes two arguments, an instance of the IACTOR class (or one of its subclasses) and the deadline of the actor. Inew creates a new instrumented actor with the specified deadline. It returns a pointer to an Imbox that can then be used like Mbox to send the messages to the instrumented actor. For example,

```
Imbox m1 = Inew (Factorial, 200)
```

creates a new Factorial actor with deadline 200 and mailbox m1. Both the Factorial actor and its mailbox (m1) are instrumented. In addition, this method also calls the update_timer method for the New operation and updates the memory variable of Resource_monitor to reflect the creation of an Mbox, a Factorial actor instance, and a stack.

4.1.2.2. Block

The estimated execution time of any sequence of C++ instructions can be specified by *block (val)*, where val is the value by which the timer should be updated to reflect the computation time for the instructions. This block of instructions does not include any of the primitive actor operations, since the time taken by these primitives is specified separately. The computation time of any assignment statement or statements involving mathematical or logical computations is specified using the block primitive.

If the execution time for a block is not precisely known at design time, the overloaded block primitive *block (val1, val2)* could be used. In this primitive, a number is randomly* selected between val1 and val2 and the timer is updated by that value.

4.1.2.3. Branching and Looping

Branching instructions in ACT++ can be specified by an *if* statement. A *while* statement is used for specifying a loop. To keep uniform syntax for if and while instructions, the user needs to create a conditional object of type Cond. The operator "!" is used to check the value of the conditional object. A conditional object returns the value true or false.

---

* The function drand, provided by UNIX system, is used to generate the random number.

4.1.2.4. If Conditional

To simulate a conditional in an instrumented system, the user gives the probability of the condition being true. The constructor for the object takes as input the probability of this object returning true. This object can then be called in a conditional as follows,

```
Cond c1 (.4)          // create conditional object c1 with probability 0.4,
if (!c1)              // c1 returns true with probability .4
{...}
```

The overloaded operator "!" invokes a method which generates a random number between 0 and 1, then compares it with the object's probability (which is .4 in this case). If the value is less, it returns true; otherwise, it returns false.

4.1.2.5. While Conditional

The while conditional can be one of two types. In the first type, two different integer arguments are specified in the constructor of the object. A randomly selected number between these two arguments is generated and stored as a private data member of the object. This number represents the loop count for the while loop. For example,

```
Cond c2 ( low, high)  // low is lower bound for loop count
                      // high is upper bound for loop count
```

```
while (!c2)          // decrement the loop count by one
{...}                // and loop until loop count is 0.
```

The overload operator "!" in this case decrements the private data member of object c2. The operator returns true if the data member is greater than zero.

In the second type, a conditional object is created with a value which is the probability of the object returning true. The second conditional type is illustrated by the following example,

```
Cond c3 ( .4)        // create conditional object c3
while (!c3)          // returns true with probability .4
{...}
```

This usage is similar to that of the if conditional.

4.1.3. Memory

ACT++ creates three different dynamic objects during an actor computation, viz. Mbox, Cbox, and behavior. Since Mbox and Cbox have a fixed size, only the number of Cboxes and Mboxes created during execution need be known to determine their total memory requirement. However, each behavior object has a different memory requirement. The size of a behavior object is equal to the size of class in which the object is an instance.

The memory variable in the Resource_monitor is always incremented to reflect the creation of new objects. However, since the memory is not infinite, there has to be some way to reclaim the memory no longer in use. This is done by an automatic garbage-collector. The design of an automatic garbage-collector is described in [Nelson 89].

4.2. Modifications to Build a Real-Time System

Two kinds of changes were required to make ACT++ suitable for simulating a real-time system. The first type of change allows a user to declare the tasks of the systems and create actors with deadlines. Next, changes were made in the worker to incorporate a priority-driven scheduler. This scheduler is based on the deadline-driven algorithm. Changes were also made to provide a pseudo-clock in the system to activate tasks based on their periods.

4.2.1. Task

There is no concept of a task provided in the ACT++ system. ACT++ code is written as a series of actor scripts coordinating among themselves. The programmer's notion of a task is a sequence of actor executions triggered by a message to a *receptionist*. A receptionist is the actor in a task that is assigned to receive a message from outside the task. To trigger a task, a message is sent to the receptionist of the task, which in turn forwards the message to the appropriate actors within the task.

As we have seen in Chapter 1, real-time task scheduling of actors is a two-step process. The first step is carried out by running instrumented scripts. This step gives an estimate of the time taken to produce the result of a single task. In the second step, the tasks are specified to a real-time scheduler. This second step is accomplished by an *add_task* method, which takes the period and worst-case computation time of the task as input and calculates the load on the system as described in Chapter 1. If the load is less than 100%, the task is added to the system. In addition to these two parameters, the add-task method also

takes the receptionist of the task as a parameter. The receptionist is declared by giving the address of one of its methods and the address of its mailbox. Thus, when a task is added to the system, the address of its receptionist is made known to the Clock object, explained later. The Clock object periodically invokes the task (depending on the period specified in the add_task method).

As explained in section 2.1.1, aperiodic tasks could be added to a periodic system using slots in which they are scheduled to run. To add an aperiodic task in our system, the user first needs to reserve a slot using the *add_slot* method. This slot could be shared by several aperiodic tasks if they are mutually exclusive (i.e., only one of the tasks among that task set could be active at any time). The method add_slot takes two parameters:

- period -- the minimum of the periods of the tasks sharing the slot,
- execution-time -- the maximum of the execution times of the tasks sharing the slot.

This step is required to calculate the load on the system and thus to determine the feasibility of the scheduling. The aperiodic tasks could then be added to the system using the overloaded add_task method.

The deadline for the receptionist is specified during its creation in the method Inew. The receptionist creates other actors which are assigned the same deadline as the receptionist. These actors in turn create other actors with the same deadline.

4.2.2. Clock

A static object is created in the real-time system to provide a simulated Clock. It is updated by the update_timer method. The Clock object has information about periods and the mailboxes of the receptionist of all the tasks in the system. It invokes a task whose period has completed by sending a

message to its receptionist. This message has no parameters and is only used to put the receptionist in the ready to run queue.

4.2.3. Extensions to Worker

As discussed in Chapter 3, the original ACT++ worker is a FIFO scheduler. It schedules the actors in the order they are added to the scheduling queue. However, to guarantee that the deadlines of the tasks are met we have chosen the deadline-driven scheduling scheme. The reasons for choosing this algorithm are that it is very simple to implement, it allows 100% processor utilization, and it is appropriate for dynamic systems. To provide scheduling priorities, the actors are added to the scheduling queue in the order of their deadlines. So an actor with an earlier deadline is put towards the head of the queue while an actor with a later deadline is put further back in the queue. The actor selected for scheduling is always retrieved from the head of the queue and the head is updated to the next element in the queue.

5. Automobile Management System

This chapter illustrates, by an example, the technique presented in Chapter 4 to estimate the execution time and memory requirement of a system. The example, the Automobile Management System (AMS), is taken from the book *Strategies for Real-Time System Specification* [Hatley 87]. This example has been used earlier as a basis for comparing several different development methods. The complete specification of the problem is given in Appendix A. The complete system and the code for it is given in Appendix B.

An Automobile Management System has the following functions:

- cruise control,
- average speed monitor,
- fuel consumption monitor, and
- maintenance monitor.

As shown in Appendix A, the system is modeled as four different actors: Control_throttle, Measure_motion, Measure_mile, and Monitor_status. Along with these, there are four input processing actors that interface with different devices. According to the information provided by these actors they are named: Brake_status, Gear_status, Engine_status, and Shaft_rotations.

In this chapter we concentrate on two tasks in the cruise control part of the AMS. One of these tasks, *task_activate*, is an aperiodic task, while the other, *task_rotation*, is a periodic task. We will write the

instrumented script for both of these tasks and create an instrumented system consisting of these two tasks. Figure 4 shows the major elements of the instrumented system. The role of these elements is described in more detail in the following subsection.

5.1. Description of the Tasks

The first step in the execution of the instrumented system is the creation of all actors using the Inew operation. These newly created actors are then sent the mailbox addresses of the actors they need to communicate with. The instrumented system then creates various tasks. In this example the tasks are task_activate and task_rotation. Creating a task means that the system creates the receptionist of the task. Whenever a command is input, a message is sent to the receptionist of the task by the User_handler function; or, in the case of periodic tasks, the message is sent by the Clock object. The design of User_handler is described in section 5.3.

The cruise control function is activated by the user by entering the command Activate. The execution of this command results in the system selecting the current speed or the default speed (which is 30 miles per hour), whichever is higher, and maintaining that speed for the car. The corresponding task, task_activate, is invoked by the system to handle the execution of the command. The message sent to the receptionist is forwarded to the Control_throttle actor. Control_throttle then sends a message to the Measure_motion actor to obtain the current speed of the car. Control_throttle then holds the throttle at a position required to maintain the speed of the car.

The task task_rotation is a periodic task that is invoked once each second by the Clock object. The Clock object sends a message to the Shaft_rotation's receptionist, which sends the number of pulses, the pulse_rate, and the rate of change of the shaft rotation to Measure_motion (these values are gathered by a device which monitors shaft rotation). To simulate the device, we generate random numbers for each of the above values. Measure_motion uses these values to calculate distance_count, speed, and acceleration

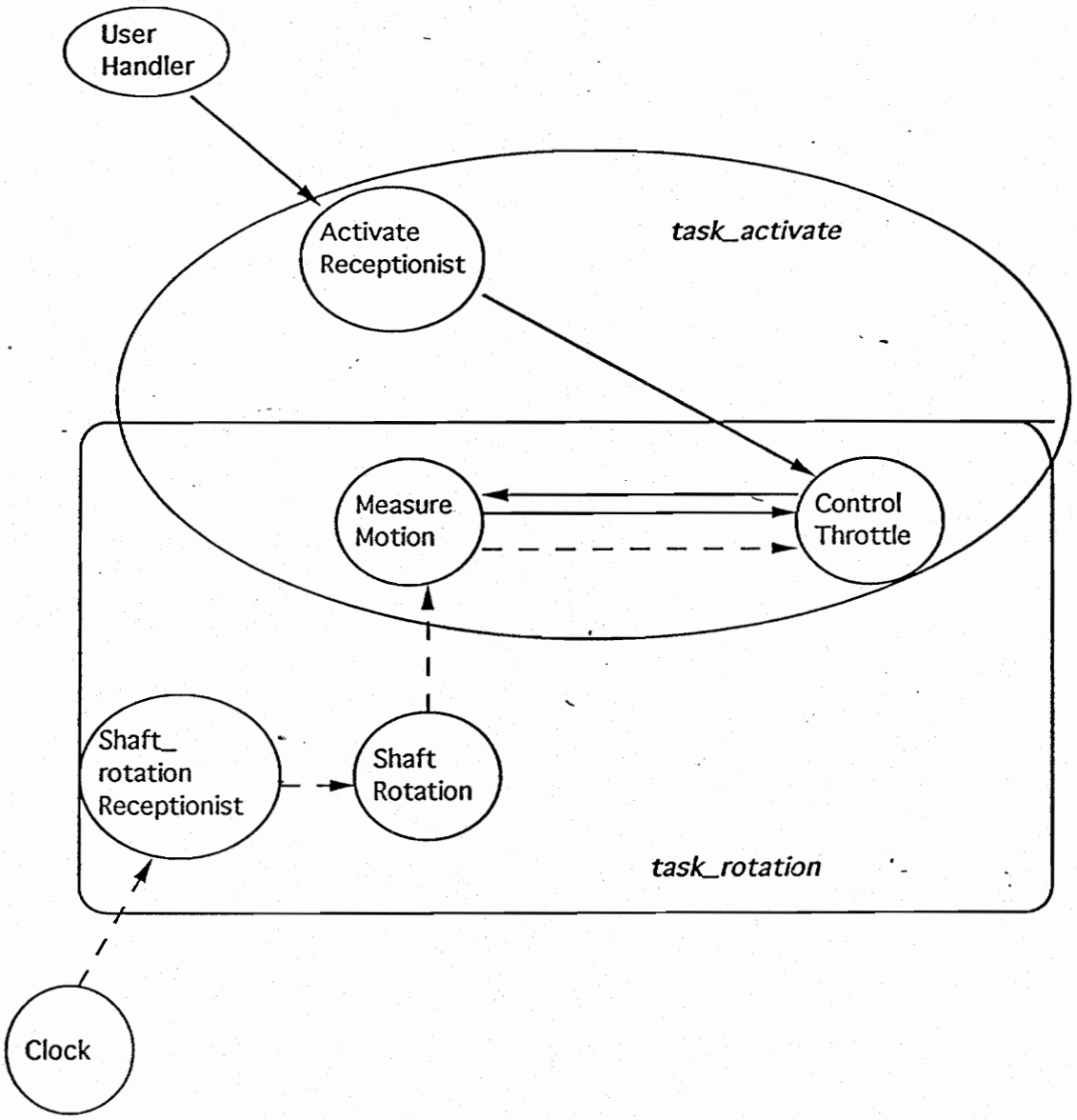


Figure 4. Task Structure of Instrumented System

of the vehicle using the formulae given in Appendix A. Measure_motion sends these values to Control_throttle, which calculates the throttle position value and sends the value to the throttle.

5.2. Writing the Instrumented Script

Information about a task's execution time and memory requirement is gathered by executing a script written with instrumentation primitives. Running this script yields the statistics about one execution of the task for given data. In certain cases, it may be desirable to run the system repeatedly with different inputs to get a better idea of average and worst-case execution times and memory requirements. However, in the AMS system, the two tasks, task_activate and task_rotation, have only one kind of input. As a result, different runs of the task will always perform the same execution steps. Since execution times and memory requirements for different runs are the same, each instrumented task is executed only once.

As shown in Figure 4, three of the actors take part in the execution of both tasks. These shared actors are: Measure_motion, Control_throttle, and Shaft_rotation. Among these, Measure_motion and Shaft_rotation are simple actors; they have only one state. However, Control_throttle is an actor with state information, and is modeled by creating different classes which provide different interfaces for it. An actor changes its interface by becoming an actor of an another class [Kafura 90].

The code for the two actors, Shaft_rotation and Measure_motion, is given in Appendix B. These actors are developed as classes Shaft and Motion. They are created as actors by actor Init before the execution of any task starts. The actor Init also sends these actors the address of mailboxes of the actors they need to communicate with. The actor Shaft_rotation is a simple actor; it periodically (every 1 second or 1000 time units in our code) gets a message from the Clock object. It then sends the incremental distance count, pulse rate, and rate of change of shaft rotation to actor Measure_motion. These values are generated by a random number generator. Measure_motion uses these numbers to calculate distance, speed, and acceleration of the vehicle using the formulae given in Appendix A (under subsection Measure_motion). The Measure_motion actor sends one of these values to Control_throttle, which in

turn controls the throttle by calculating the throttle position using the formulae given in Appendix B (under the subsection Maintain Speed and Maintain Acceleration).

The state diagram for the actor Control_throttle, which is an instance of the class Throttle, is shown in Appendix A (Figure 5). Different states in this diagram are represented by subclasses of the class Throttle. For example, when Control_throttle is in the state idle it becomes an actor of class throt_idle. Similarly, when Control_throttle is in the state cruising or acceleration, it becomes an actor of subclass throt_crus or throt_accl, respectively. Each of these three classes -- throt_idle, throt_crus, and throt_accl -- are subclasses of class Throttle. The class Throttle provides the methods maint_accl, maint_speed, and select_speed (to maintain acceleration, maintain speed, and select_speed as shown in Figure 5). The code for these methods follows the formulae given in Appendix A (under the same names).

To gather statistics, main first invokes the method stat of the actor Init. Actor Init calls two functions, task_activate_recep1 and task_rot_recep1. The code for the functions is given in the file fun1.c. The task_rot_recep1 method:

- sets the timer variable,
- creates the receptionist for the task task_rotation,
- invokes task_rotation by sending a message to its receptionist. The execution further follows the path shown by dotted lines in Figure 4 (instead of Clock, the receptionist in this case receives a message from the function task_rot_recep1).
- Prints the statistics for this execution (values for this run indicate a memory requirement of 56 units and an execution time of 90 units).

The method task_activate_recep1 creates the receptionist for the task task_activate. The execution in this case follows the path shown by solid lines in Figure 4 (here the function starts the execution instead of User_handler).

5.3. Building the System

The first step in building a real-time system is to determine scheduling feasibility. This analysis is carried out each time a task is added to the system. The aperiodic task, task_activate, is added to the system first, using the add_slot primitive to assign a slot to the task. The function task_activate_recep assigns the slot for the task task_activate. This function then sends its receptionist's mailbox address to User_handler. This function also sends the mailbox address of the throttle. The task task_rotation is added using the function add_task in method init of actor Init. The method task_rot_recep is invoked in this call and returns the mailbox address of the receptionist of the task. An arbitrary value of 1000 units both in add_slot and add_task is the period of these tasks. The only restriction on the period of the task is that the load should be less than 100%. The period of 1000 units simplifies the calculation and ensures the correct load requirement.

To create the AMS instrumented system, we need to create a User_handler that gives the commands to the system to execute different tasks. We can provide input to the system by one of following techniques:

- Random generated input: user input can be generated randomly (i.e., activate the tasks in some random order). Since there are precedence relations among various tasks, inconsistent input may be generated which will result in the system either not responding to the input or producing incorrect results. For example, braking, followed by stop acceleration, will be inconsistent and the system will give the wrong results.

- Script file: The user can write input data in a script file for the system to read. To obtain long-term performance measures, a large number of inputs may need to be specified. As a result, the file could be very large.

- User generated: The user can be prompted for different inputs, and depending on the input, the appropriate tasks can be activated. The problem with this scheme is that we depend on the user to provide input consistent with the state of the system. This means that the user

has to be aware of the current state of the system. However, we choose this scheme for its simplicity and because it closely models the real problem, where input will be user driven.

The actor User_handler is created in the beginning along with other actors. User_handler is given the addresses of mailboxes of the receptionists of various tasks through various methods provided by the User_handler actor. The User_handler actor is initially activated by the system by sending it a message. This step is performed in function init of actor Init and is done after all the tasks have been created. The User_handler, when activated, will wait for an input from the user. On receiving the input, it will send the appropriate message to the receptionist of the task and send itself a message so that it (User_handler) is put back in the ready to run queue.

6. Conclusions and Future Research

This report described the design of instrumentation primitives. These primitives can be used by a programmer to determine the resource requirements of a real-time system written in an actor-based language. It outlined the two-stage process of developing the real-time system and showed an example of a system developed by using this technique.

One of the initial goals of the project was to keep to a minimum the difference between a program written in ACT++ and the instrumentation scripts. This was accomplished by providing the same syntax for creation of objects and the operations performed on them. This allowed an Mbox to be created using the same syntax in both ACT++ and the instrumentation scripts. This was a logical goal, since object-oriented languages provide us with the ability to define virtual functions [Stroustrup 87]. This allows objects of various types to be treated in a uniform manner. Thus, the operator << is defined as a virtual function both in classes Mbox and Imbox. The C++ object can determine for itself which operation it needs to perform, which should have allowed us to use instrumented objects and ACT++ objects in same way (since instrumented objects are subclasses of ACT++ objects). That way an instrumented program would have looked exactly like an ACT++ program (except where the word ACTOR occurs it needs to be changed to IACTOR). However, we could not use this property in our system because of the way ACT++ objects are created (and ACT++ objects have to be created that way because of certain other limitations imposed by the C++ compiler). These limitations could be removed if a compiler for ACT++ is developed.

Still, there were several advantages in choosing an object-oriented "language" like ACT++ to develop the instrumentation primitives:

- It is very easy to implement. Most of the classes are defined as subclasses to already existing class definitions in ACT++ classes.

- A program written in ACT++ and the instrumentation scripts look very much the same except for minor changes involved in creating the ACT++ objects. This helps in shifting from a specification script to a complete ACT++ program.

6.1. Future Work

The primitives can be extended to provide debugging and visualization information. This can be done by providing additional methods in the instrumented classes. Whenever an operation is performed on an instrumented object, these methods could be activated to capture the required information. This allows a computation to be traced or display all of the messages received by a particular Imbox.

The earliest-deadline-first algorithm performs poorly because, when calculating the load on the system, it uses worst-case analyses. In certain cases the differences between the results of average-case and worst-case analyses are very large. A system designed with these results could perform very poorly. To improve the efficiency (load), we could associate a *penalty* for missing deadlines. Tasks will have both a deadline and a penalty. When conflict arises, a task could be chosen for scheduling that has the highest penalty of missing the deadline associated with it.

To meet performance and reliability goals, real-time systems are often implemented as a network of loosely coupled processors. However, using Mok's argument shown in chapter 3, we can prove that a dynamic multiprocessor system will be non-optimal, unless the scheduling is centralized. In actor-based systems, tasks consist of various actors which can be independently scheduled to run on different processors. Thus, further work is needed on "internal scheduling," which schedules the various actors of a task.

Since the example system that we have chosen does not have much variation, the simulation results tend to stabilize very quickly. We can arbitrarily execute the system 100 times and get the average and worst-case results for execution times. However, in certain complicated systems, we should use better simulation techniques to determine when to terminate the simulation runs.

7. Bibliography

- [Agha 86] Gul Agha, *A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.
- [Davari 86] S. Davari and S. K. Dhall, "An On Line Algorithm for Real-Time Task Allocation," *In:* IEEE Real-Time Systems Symposium, December 1986.
- [Hatley 87] Derek J. Hatley and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House Publishing, 1987.
- [Hewitt 76] Carl Hewitt, "Viewing Control Structure as Patterns of Passing Messages," *In:* AI MEMO 410, MIT Artificial Intelligence Laboratory, 1976.
- [Kafura 88] Dennis Kafura, "Concurrent Object-Oriented Real-Time System Research," *In:* Technical Report, TR 88-47, Department of Computer Science, Virginia Polytechnic Institute and State University, 1988.
- [Kafura 90] Dennis Kafura and Keung Hae Lee, "Building a Concurrent C++ with Actors," *In:* JOOP, May 1990, Pages 25-37.
- [Liu 73] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *In:* Journal of Association for Computing Machines, February 1973, Pages 46-61.
- [Mok 84] Aloysius K. Mok, "The Design of Real-Time Programming Systems Based on Process Model," *In:* IEEE Real-Time Systems Symposium, 1984, Pages 5-17.
- [Nelson 89] Jeffrey Nelson, Automatic, Incremental, On-the-fly Garbage Collection of Actors, MS Thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, 1988.
- [Sha 86] Lui Sha, John P. Lehoczky, Ragunathan Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Sheduling," *In:* IEEE Real-Time Systems Symposium, 1986, Pages 181-191.

- [Stankovic 88] John A. Stankovic and Sheng-Chang Cheng, "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey," *In*: Tutorial -- Hard Real-Time Systems, Computer Society of the IEEE, 1988, Pages 150-173.

- [Stoustrap 87] Bjarne Stroustrap, *The C++ Programming Language,* Addison-Wesley Publishing Company, July 1987.

- [Ullman 75] J. D. Ullman, "np-Complete Scheduling Problem," *In*: Journal of Computing and Systems Science, 10, 1975.

- [Woodbury 86] Michael H. Woodbury, "Analysis of the Execution Time of Real-Time Tasks," *In*: IEEE Real-Time Systems Symposium, 1986, Pages 89-95.

Appendix A. Problem Statement

The following appendix has been extracted from the book -- *Strategies for Real-Time Systems* (pages 295-352) [Hatley 67].

An Automobile Management System is to be developed as an optional extra on some car models. The system is to take care of several routine operations and maintenance tasks, including:

- cruise control,
- average speed monitoring,
- fuel consumption monitoring, and
- maintenance monitoring.

A description of these tasks follows. The description represents that which might be provided by the automobile manufacturer to an outside developer, and includes a requirements statement and a system-level configuration.

1. Cruise Control

The cruise control function is to take over the task of maintaining a constant speed when commanded to do so by the driver. The driver must be able to enter several commands, including: Activate, Deactivate, Start Accelerating, Stop Accelerating, and Resume. The cruise control function can be operated any time the engine is running and the transmission is in top gear. When the driver presses Activate, the system selects the current speed, but only if it is at least 30 miles per hour, and holds the car at that speed. Deactivate returns the control back to the driver regardless of any other commands. Start Accelerating causes the system to accelerate the car at a comfortable rate until Stop acceleration occurs, when the system holds the car at that speed. Resume causes the system to return the car to the speed selected prior to braking or gear shifting.

The driver must be able to increase speed at any time by depressing the accelerator pedal, or reduce the speed by depressing the brake pedal. Thus the driver may go faster than the cruise control setting simply by depressing the accelerator pedal far enough. When the pedal is released, the system will regain control. Any time the brake pedal is depressed, or the transmission shifts out of top gear, the system must go inactive. Following this, when the brake is released, the transmission is back in top gear, and Resume is pressed, the system returns the car to the previously selected speed. However, if Deactivation has occurred in the intervening time, Resume does nothing.

Since speed and distance per drive shaft rotation are affected by tire size and wear, the system is to have a calibrated capability. For this purpose, two further controls will be provided: Start Measured Mile and Stop Measured Mile. They are only effective when the cruise control is inactive. The driver presses Start Measured Mile at the start of a measured mile, drives the mile, and then presses Stop Measured Mile. The system is to record the number of shaft pulses over this interval, and use this count as its mile reference in all calculations.

2. Average Speed Monitoring

The system is to provide the driver with an average speed indication on its display. The driver enters a Start Trip command, and later, whenever he enters an Average Speed request, the system will display the average speed of the car for all the time that the engine has been running since the Start Trip. This continues till next time the Start Trip command is used.

3. Fuel Consumption Monitoring

Whenever the tank is filled, the driver may enter the quantity of fuel added since the last fill-up. The system will then calculate and display the fuel consumption over that period.

4. Maintenance Monitoring

The system will monitor the car's mileage, and notify the driver of required maintenance according to the following schedule:

Oil and oil filter change	5,000 miles
Air filter change	10,000 miles
Major service	15,000 miles

Two hundred and fifty miles before each required service, the appropriate message will appear intermittently: fifty miles before, it will appear continuously, and will remain until the driver enters a Service Complete command.

5. Requirements and Architecture Development

The requirements and architecture model that follows reflects the requirements and architecture decisions that typically would be made in the course of discussions with the user and customer. Some of these resolutions are listed below.

- The system controls the car through an actuator attached to the throttle. This actuator is mechanically in parallel with the accelerator pedal mechanism, such that whichever one is demanding greater speed controls the throttle. The system is to drive the actuator by means of an electrical signal having a linear relationship with throttle deflection, with 0 volts setting the throttle closed and 8 volts setting it fully open.

- The system is to measure speed by counting pulses it receives from a sensor on the drive shaft. Count-rate from this sensor corresponds to the vehicle's miles per hour through a proportionality constant.

- When the system senses that the speed is more than 2 miles per hour above the selected speed, it is to completely release the throttle (this situation will occur when driving downhill). At any speed below this, it is to drive the throttle to a deflection proportional to the speed error until, at 2 miles per hour below the selected speed, the throttle is fully open (the steep downhill situation). The system thus serves as feedback or control part of a servo loop, in which the engine is the feed-forward part. For smooth and stable servo operation, the system must update its output at least once per second.

- To avoid rapid increases in acceleration, the actuator must never open faster than to traverse its full range in 10 seconds. It may close at any rate, however, since the car just coasts

when the throttle is closed. The automotive engineers have determined that, with these characteristics, the system will hold the car within 1 miles per hour of the selected speed on normal gradients, and will give a smooth, comfortable ride.

- When the system is accelerating the car, it must measure the acceleration and hold it at 1 miles per hour/sec, and the throttle should be closed; at 0.8 miles per hour/sec, it should be fully open. Between these limits, the opening is to be linearly related to acceleration.

- The automobile system will have a fully numerical keyboard and a CRT display.

- The user interface module for the system will be "intelligent," such that all driver entries will be validated by checking for reasonableness (for example, range on fuel tank input or other appropriate measures for other inputs).

6. Requirements Model

The complete requirements model follows this section. Since this is a complete model, the diagram and specification numbers are sufficient to identify them, and figure numbers are not used. This system must interface with several other systems in the car and with the driver. This is illustrated in the context diagrams in which our system is shown performing the task Control and Monitor Auto, and interfacing with the driver, driver shaft, transmission, engine, brake, and throttle mechanism.

Figure 6 and Figure 7 show that we chose to divide the model into four major processes: Measure_motion, Measure Mile, Control Throttle, and Monitor Status. Again, these correspond quite closely to the problem statement. Each of them is decomposed down another level or two.

7. Formulae

7.1 Measure_motion

For each pulse of the Shaft Rotation:

add 1 to Distance_count then set:

Distance = Distance_count / Mile_count

At least one per second, measure pulse rate of Shaft Rotation in pulses per hour, and set:

Speed = pulse rate / Mile_count

At least once per second, measure rate of change of Shaft Rotation pulses in pulses per hour per seconds, and set:

Acceleration = Rate of change / Mile_count

7.2 Select Speed

Issue Desired Speed = Speed

7.3 Maintain Speed

Set:

$$V_{Th} = \begin{cases} 0 & (S_D - S_A) > 2 \\ 2(S_D - S_A + 2) & -2 <= (S_D - S_A) <= 2 \\ 8 & -2 > (S_D - S_A) \end{cases}$$

subject to:

$dV_{Th}/dt <= .8V/sec.$

where:

V_{Th} = Throttle Position

S_D = Desired Speed

S_A = Speed

7.4 Maintain Accel

Set:

$$V_{Th} = \begin{cases} 0 & 1.2 > A \\ 20\ (1.2 - A) & 0.8 <= A <= 1.2 \\ 8 & 0.8 > A \end{cases}$$

subject to:

$dV_{Th}/dt <= .8V/sec.$

where:

V_{Th} = Throttle Position

A = Acceleration

7.5 Issue Average Speed

Issue:

Average Speed = (Distance - Start Distance) / Trip Time

7.6 Monitor Fuel Consumption

Issue:

Fuel Consumption = (Distance - Refuel Distance) / Fuel Qty

The rest of the appendix contains four figures from the book. The first one illustrates the state diagram for the Control_throttle. The second and the third figures show the control flow and the data flow in the Automobile Management System. Finally, Figure 8 contains the names of all the tasks and their deadlines.

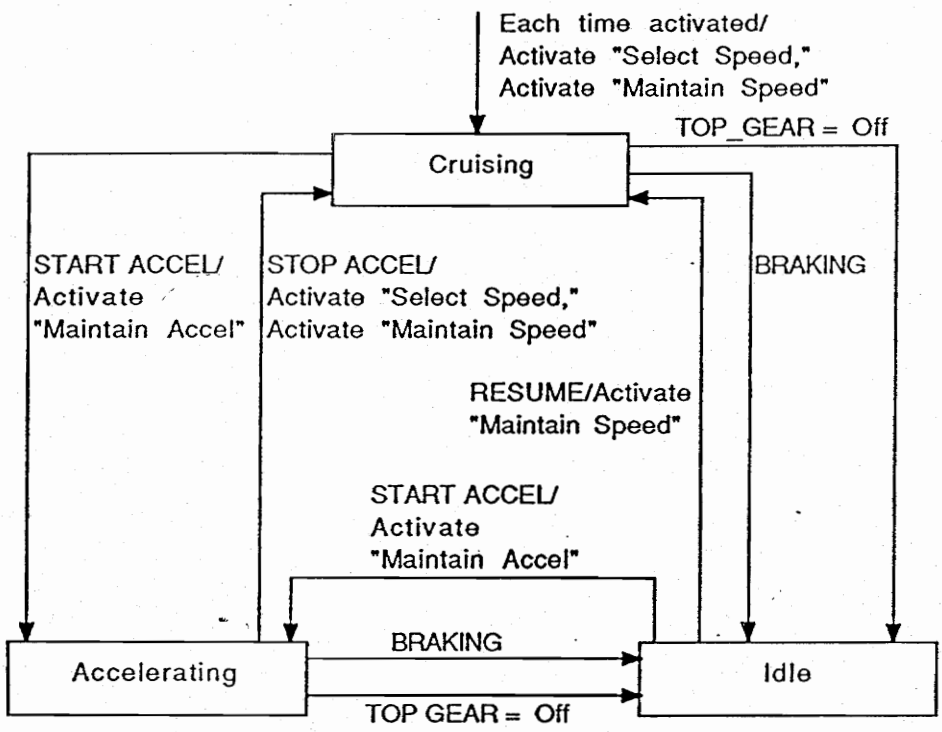


Figure 5. State Diagram for Control_throttle

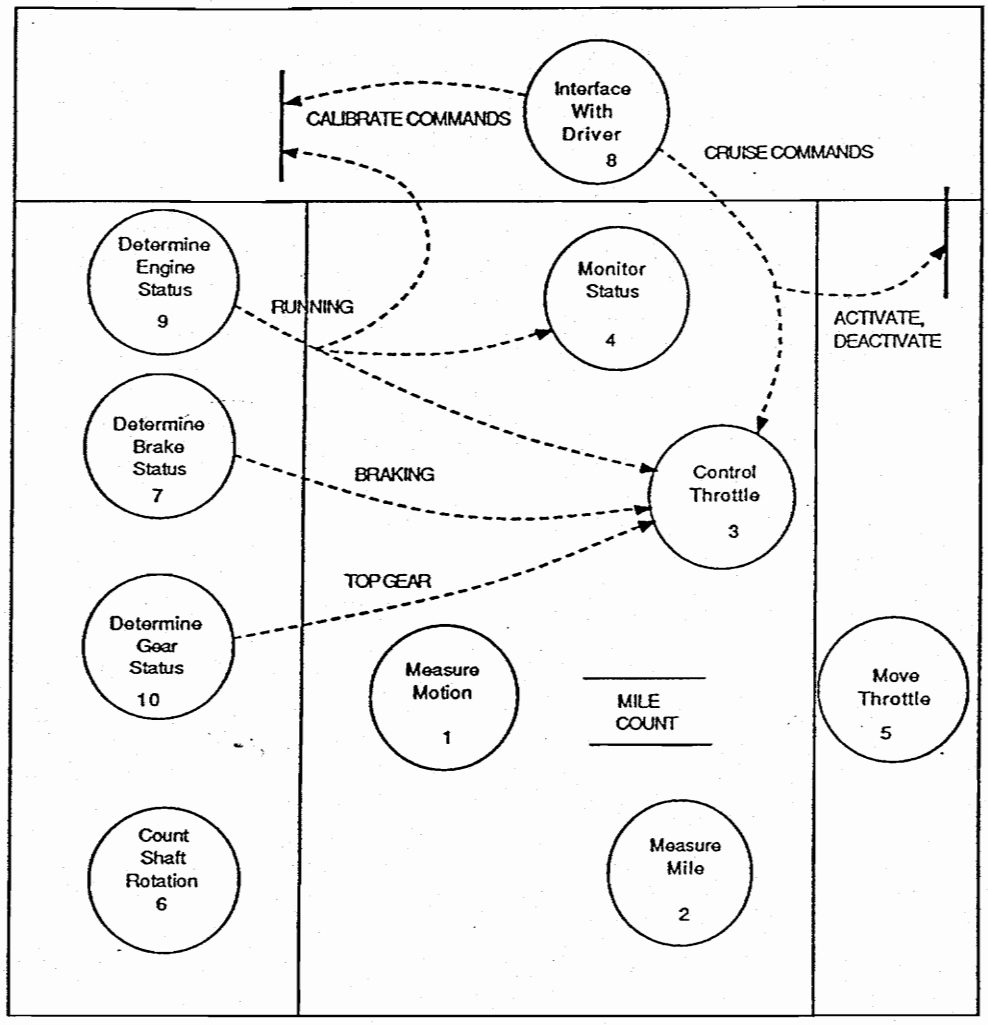


Figure 6. Control Diagrams for AMS

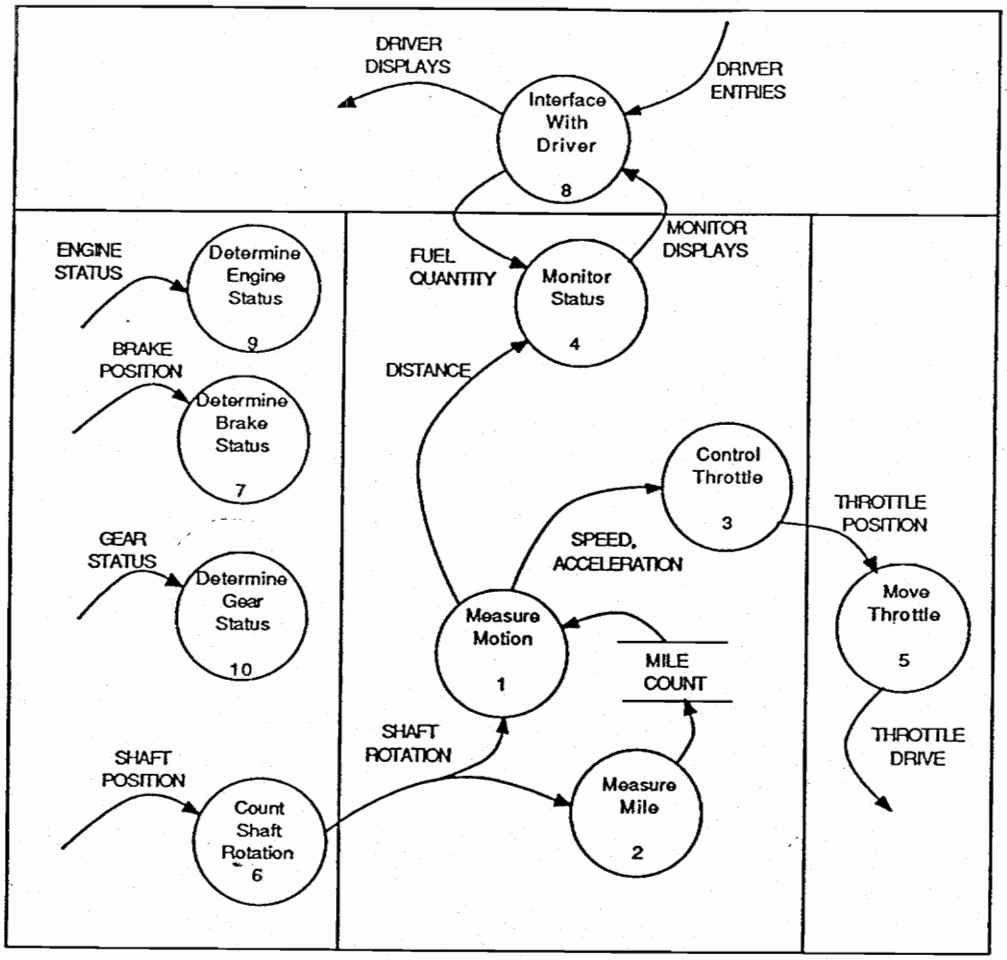


Figure 7. Data Flow Diagram for AMS

INPUT	EVENT	OUTPUT	EVENT	RESPONSE TIME
ACTIVATE	Turns On	THROTTLE POSITION	Goes to selected cruise value	0.5 sec. max.
RESUME	Turns On			
DEACTIVATE	Turns On	THROTTLE POSITION	Goes to null condition	0.5 sec. max.
TOP GEAR	Turns Off			
BRAKING	Turns On			
START ACCEL	Turns On	THROTTLE POSITION	Goes to acceleration value	0.5 sec. max.
STOP ACCEL	Turns On while THROTTLE POSITION at acceleration value	THROTTLE POSITION	Goes to cruise value	0.5 sec. max.
RUNNING	Turns On	THROTTLE POSITION	Goes to null value	0.5 sec. max.
SHAFT ROTATION	Rotation rate changes	THROTTLE POSITION	Corresponding change in value	1 sec. max.
	Distance since last maintenance exceeds maintenance warning distance	MAINT NEEDED	Displayed	10 sec. max.
FUEL QUANTITY	Entered	FUEL CONSUMPTION	Displayed	1 sec. max.
AV SPEED RQST	Turns On	AV SPEED	Displayed	1 sec. max.
START TRIP				No time-critical outputs
START MEASURED MILE				
STOP MEASURED MILE				

Figure 8. Timing Specification for Different Tasks

Appendix B. Code for the Example

There are two main functions in here. The first one is used in calculating the statistics of the tasks. This invokes the function stat of the actor Init. The function stat calls functions task_activate_recep1 and task_rot_recep1 to create the receptionists and gather the statistics for the tasks. These two functions are shown in file fun1.c. In second case the main calls the function init of actor Init. Here init calls the functions task_activate_recep and task_rot_recep to create the system. The code for these functions is given in file fun2.c. The code for both the main functions is in only one file except the one line which is put in comment depending on which case is run.

```
/* itask.c */
#include "iinst.h"

/* this main is for the purpose of gathering the statistics of the execution of
   the tasks. It calls stat function of actor Init */

    main(){
    Cbox cbox;
    printf("starting main\n");

    Mbox ini = Inew (Init,5); // create init function to start the work
```

```
/*this line for collecting stats */
      ini <= Mail(&Init::stat,cbox);

/*this line for system */
//       ini <= Mail(&Init::init,cbox);


      printf("calling receptionist\n");
      ~cbox;
      printf("END \n");

      }
```

```
/*************************************************************************/
/*
/*    Class Name          -       Init
/*
/*    File Names -        init.h, init.c
/*
/*    PROGRAMMER    -     SANJAY KOHLI
/*
/*    DESCRIPTION   -     This class is responsible for creating
/*                        all the actors and calling functions to
/*                        create task receptionists.
/*
/*    DATE      -         Sept 25, 1989.
/*
/*************************************************************************/



/* init.h */

class Init : public Iactor {
public:
      void init();
      void stat();
      };

/* init.c */

 /*This function is called when statstics are measured, it is called by main
this function is responsible for starting the system.  It creates all the actors
before    starting the tasks.  Then it creates the receptionists for all the
tasks and the    initiate the User_handler to start the work.  User_handler takes
the input from the    user and initaite the tasks */


void Init::stat() {

      Imbox& mbox_motion = Inew (Motion,5); //create the actor Measure_motion
      Imbox& mbox_data = Inew (Data,5); //create Data
      Imbox& mbox_throt = Inew (Throttle,5); //create throttle
      Imbox& mbox_shaft = Inew (Shaft,5); //create Shaft_rotation
      Imbox& mbox_mile = Inew (Mile,5); //create Measure_mile
      Imbox& mbox_user = Inew (User,5); //create the User_handler

      Icbox cbox;
```

```
        //following statements sends the initial mailbox ids of all the
        //actors just created.  They are send and the function init of these
        //actors is invoked which saves these ids as private variables.


        mbox_throt <= Mail(&Throttle::init) << (int)&mbox_motion;
        mbox_motion <= Mail(&Motion::init) << (int)&mbox_data <<
                (int)&mbox_throt << (int)&mbox_shaft;
        mbox_shaft <= Mail(&Shaft::init,cbox) << (int)&mbox_motion;

        ~cbox;

        // Call function to create task activate
        task_activate_recep1((int)&mbox_throt, &mbox_user);

        //Call function to create task shaft_rotation
        task_rot_recep1((int)&mbox_shaft, &mbox_user);


        Reply(1);
        }

/* this actor is responsible for starting the system.  It creates all the actors
before    starting the tasks.  Then it creates the receptionists for all the
tasks and the    initiate the User_handler to start the work.  User_handler takes
the input from the    user and initaite the tasks */


void Init::init() {

        Imbox& mbox_motion = Inew (Motion,5); //create the actor Measure_motion
        Imbox& mbox_data = Inew (Data,5); //create Data
        Imbox& mbox_throt = Inew (Throttle,5); //create throttle
        Imbox& mbox_shaft = Inew (Shaft,5); //create Shaft_rotation
        Imbox& mbox_mile = Inew (Mile,5); //create Measure_mile
        Imbox& mbox_user = Inew (User,5); //create the User_handler


        Icbox cbox;

        //following statements sends the initial mailbox ids of all the
        //actors just created.  They are send and the function init of these
        //actors is invoked which saves these ids as private variables.

        mbox_throt <= Mail(&Throttle::init) << (int)&mbox_motion;
        mbox_motion <= Mail(&Motion::init) << (int)&mbox_data <<
                (int)&mbox_throt << (int)&mbox_shaft;
```

```
mbox_shaft <= Mail(&Shaft::init,cbox) << (int)&mbox_motion;

~cbox;

// Call function to create task activate
task_activate_recep((int)&mbox_throt, &mbox_user);

//add the task shaft rotation to the system by ading it to clock
clock.add_task(1000, 50,&task_rot::recep,
      task_rot_recep((int)&mbox_shaft,&mbox_user));

 mbox_user <= Mail(&User::recep, cbox);
 ~cbox;

Reply(1);
}
```

```
/*************************************************************************/
/*
/*    File Names -          iinst.h, fun1.c, fun2.c
/*
/*    PROGRAMMER    -       SANJAY KOHLI
/*
/*    DESCRIPTION   -       These files contains: iinst.h - includes all the files
/*                          required for compilation. fun1.c contains the functions for
/*                          creating receptionists when statistics are gathered. fun2.c contains
/*                          corresponding functions for system.
/*
/*    DATE          -     Sept 25, 1989.
/*
/*************************************************************************/

/* inst.h */


#include <math.h>

// files of instrumented system
 #include "../act++/act.h"



#include "../def.h"
#include "../cond.h"
#include "../stat.h"
// #include "../clock.h"

/* create the timer and clock object */

static Stat timer;
// static Inst clock;

//#include "../iactor.h"
#include "../imbox.h"
#include "../icbox.h"
#include "../clock.h"
 static Inst clock;

/* include C files of instrumented system */

#include "../cond.c"
#include "../imbox.c"
#include "../icbox.c"
```

```
#include "../clock.c"


#include "example.h"
#include "mile.h"
#include "shaft.h"
#include "data.h"
#include "motion.h"
#include "throttle.h"
#include "activate.h"
#include "init.h"
#include "rot.h"
#include "user.h"


/* C files of the code */

#include "shaft.c"
#include "motion.c"
#include "throttle.c"
#include "activate.c"
#include "rot.c"
#include "user.c"
#include "fun1.c"
#include "fun2.c"
#include "init.c"


/* fun1.c */

/* This file contains the functions which are called by stat function
   of Init. It contain functions to create the tasks receptionists and
   is used for the cases when statistics are gathered */


/* this function is responsible for creatin the receptionsit for the task
   shaft_rotation. It also sets the timer to null and then try to print
   the stats after the execution */


void task_rot_recep1(int mbox_shaft, Imbox* mbox_user)
{
      Icbox cbox;

      // create the receptionist for task shaft_rot
      Imbox& rcp = Inew( task_rot,5);
```

```
    timer.reset(); // reset the timer for this task

    rcp <= Mail(&task_rot::recep1,cbox) << mbox_shaft;

    ~cbox;

    printf("This statastics are for task rotation\n");
    timer.print_stat(); // print the`stats for this task
    }


/* this function is responsible for creatin the receptionsit for the task
   actvate. It also sets the timer to null and then try to print
   the stats after the execution */


void task_activate_recep1(int mbox_throt, Imbox* mbox_user)
{
    Icbox cbox;
    //create receptionist for task_activate
    Imbox& rcp = Inew( task_activate,5);

    *mbox_user <= Mail(&User::update_active) << (int)&rcp;
    timer.reset(); // set timer values so to take values for this guy

    rcp <= Mail(&task_activate::recep1, cbox) << mbox_throt;

    ~cbox;

    printf("Following are the statistics for task_activate \n");
    timer.print_statf(); // print the statisctics for this task
    }



/* fun2.c */

/* this file contains the functions which are involved in creating the
receptionists for the tasks in running simulated system */


/* this function is called by init, it creates the task activate reception
   and updates the User_handler with its id. It also adds the task in system */

Imbox& task_rot_recep(int mbox_shaft, Imbox* mbox_user)
{
```

```
    Icbox cbox;

    // create the receptionist for task shaft_rot
    Imbox& rcp = Inew( task_rot,5);
    *mbox_user <= Mail(&User::update_rot) << (int)&rcp;


    rcp <= Mail(&task_rot::update_mbox,cbox) << mbox_shaft;

    ~cbox;

    return rcp;
    }

/* this function is called by init, it creates the task activate reception
  and updates the User_handler with its id. It also adds the task in system */

void task_activate_recep(int mbox_throt, Imbox* mbox_user)
{
    Icbox cbox;

    //create receptionist for task_activate
    Imbox& rcp = Inew( task_activate,5);

    //also add the slot for this task

    clock.add_slot(1000, 60); // 1000 is arbit figure

    *mbox_user <= Mail(&User::update_active) << (int)&rcp;

    rcp <= Mail(&task_activate::update_mbox,cbox) << mbox_throt;

    printf("Following is the task_activate \n");
    ~cbox;

    }
```

```
/*************************************************************************/
/*
/*    Class Name          -        Motion
/*
/*    File Names -        Motion.h, Motion.c
/*
/*    PROGRAMMER    -        SANJAY KOHLI
/*
/*    DESCRIPTION     -        This class contains the definition of object
/*                             Measure_motion.
/*
/*    DATE      -         Sept 25, 1989.
/*
/*************************************************************************/


/* Motion.h */

class Motion : public Iactor {

      Imbox* mbox_shaft; // mbox of shaft
      Imbox* mbox_mile_count; //mbox of Data
      Imbox* mbox_throttle; //mbox of throttle
      int dist_count; //dist_count
      int state; //current state of throttle
      int speed; //current speed of vehicle
public:

      Motion(int );
      ~Motion() { };
      void init();
      void update();
      void update_throttle();
      void update_throt();
      void reply_speed();
      };

/* to create the object with a deadline */

Motion::Motion(int dead) : (dead)

{
      speed = 20; // start with some arbit value of speed and Distance
      dist_count = 200;
      state = DEACTIVATE; //inital state of throttle
      }
```

```
/* Motion.c */

/* this function update the private variables of the actor Motion. these values
are the    mail addresses of various actors and are send as integer addresses,
so they need to be    typecasted here. */

void Motion::init()
{
      int temp1, temp2, temp3;

      Msg >> temp1 >> temp2 >> temp3;

      mbox_mile_count = (Imbox*) temp2;
      mbox_throttle = (Imbox*)temp3;
      mbox_shaft = (Imbox*) temp1;
      }

/* this function is updated by shaft rotation, and gets these values. Depending
  on the current state of Throttle it sends appropriate value to throttle */

void Motion::update()
{
      Icbox cbox_mile;
      int mile;
      int pulse, pulse_rate, rate_change;

      Msg >> pulse >> pulse_rate >> rate_change;

      dist_count += pulse;

      *mbox_mile_count <= Mail(&Data::Mile_count, cbox_mile);
      ~cbox_mile >> mile;

      // no message in case of idle or deactivate

      if(state == CRUS) {
            *mbox_throttle <= Mail(&throt_crus::update,cbox_mile)
                              << (pulse_rate / mile);
            ~cbox_mile; // To get statistics only
            speed = pulse_rate / mile;
            }
      if (state == ACCL) {
        *mbox_throttle <= Mail(&throt_accl::update) << (rate_change /mile);
            }

      Reply(1); // because shaft would be waiting while calculating stat
```

```
    }

void Motion::update_throttle()
{

    Icbox cbox_pulse_rate, cbox_mile_count;
    int pulse_rate, mile_count;

    state = CRUS;

    *mbox_shaft <= Mail(&Shaft::update_motion, cbox_pulse_rate);
    *mbox_mile_count <= Mail(&Data::Mile_count, cbox_mile_count);

    ~cbox_pulse_rate >> pulse_rate;
    ~cbox_mile_count >> mile_count;

    Reply(pulse_rate/ mile_count);
    }

/* this function is invoked by throttle to tell its latest state */
void Motion::update_throt()
{

    Msg >> state;
    }


/* this function just sends the value of current speed to throttle, and it also
knows    that throttle is in activate, it is to avoid sending two messages from
throttle */

void Motion::reply_speed()
{
    state = CRUS;
    Reply(speed);
    }
```

```
/*************************************************************************/
/*
/*    Class Name          -         Shaft
/*
/*    File Names -        shaft.h, shaft.c
/*
/*    PROGRAMMER    -     SANJAY KOHLI
/*
/*    DESCRIPTION   -     This class contains the definition of object
/*                        shaft.
/*
/*    DATE      -         Sept 25, 1989.
/*
/*************************************************************************/


/* Shaft.h */

class Shaft : public Iactor {
      Imbox* mbox_motion; //Mbox id of motion
      Imbox* mbox_mile; //mbox id of mile not used presently
public:
      ~Shaft() { }
      void init() ;
      void update();
      void update_motion();
      };

/* Shaft.c */

/* this function update the private variables of the actor Shaft. these values
are the    mail addresses of various actors and are send as integer addresses,
so they need to be    typecasted here. */

void Shaft::init()
{
      Cbox cbox;
      int motion, mile;

      Msg >> motion ;
      mbox_motion =  (Imbox*)motion;
      Reply(1);
      }

/* this function is invoked periodically to send the pulses, pulse_rate and
rate_change    to the measure_motion. it is presently generated by random
```

```
numbers */

void Shaft::update()
{
      Cbox cbox;

      *mbox_motion <= Mail(&Motion::update,cbox) << randint(PULSE) <<
                  randint(PULSE_RATE) << randint(RATE_CHANGE);
      //mbox_mile <= Mail(&Measure_mile::update) << randint(PULSE_COUNT);

      ~cbox;
      Reply(1);

      }

void Shaft::update_motion() {

      Reply(randint(PULSE_RATE));
      }
```

```
/*************************************************************************/
/*
/*    Class Name          -       Throttle
/*
/*    File Names -        throttle.c,throttle.h
/*
/*    PROGRAMMER    -     SANJAY KOHLI
/*
/*    DESCRIPTION   -     This class contains the definition of object
/*                        Throttle. This object is active only in the begining
/*                        after it receves the mbox id of motion. It waits
/*                        for input handler to invoke function activate.
/*                        after which it becmes an object of type one of its
/*                        subtype.
/*
/*    DATE      -         Sept 25, 1989.
/*
/*************************************************************************/

/* throttle.h */

class Throttle : public Iactor {

public:
      int desired_speed;
      int throttle; //throttle value, at present integer later it should be a
                        //message
      Imbox* mbox_motion; //Mbox of Motion
      Throttle(int);
      ~Throttle() { }
      void init() ;
      void maint_accl(float);
      void maint_speed(int);
      void select_speed();
      void activate();
      };

Throttle::Throttle(int time) : (time)
{ } //null constructor

/* following three are subclasses of throttle and use the functions defined by
 throttle , after intial message throttle changes its state between one of these
*/

class throt_idle : public Throttle {
```

```
public:
        throt_idle(int time, Imbox* mbox, int speed, int throt) : (time)
                {
                mbox_motion = mbox;
                desired_speed = speed ;
                throttle = throt;
                }
        void update();
        void start_accl();
        void resume();
        };

class throt_accl : public Throttle {

public:
        throt_accl(int time, Imbox* mbox, int speed, int throt): (time)
                        {
                        mbox_motion = mbox;
                        desired_speed = speed ;
                        throttle = throt;
                        }
        ~throt_accl() { }
        void update();
        void stop_accl();
        void braking();
        void top_gear();
        };

class throt_crus : public Throttle {

public:
        throt_crus(int time, Imbox* mbox, int speed, int throt) : (time)
                        {
                        mbox_motion = mbox;
                        desired_speed = speed ;
                        throttle = throt;
                        }
        ~throt_crus() { }
        void update();
        void top_gear();
        void braking();
        void start_accl();
        void activate();
        };

/* throttle.c */
```

```
/* this function update the private variables of the actor throttle. these
values are    mail addresses of various actors and are send as integer
addresses, so they need to be    typecasted here. */

void Throttle::init()
{
      int tmp; // to recv the message

      Msg >> tmp; // the message is actully mailbox address of Measure_motion
      mbox_motion = (Imbox*) tmp;
      }

/* this maintains accelration by the formula given in appendix   and set
throttle */

void Throttle::maint_accl(float accl)
{
      Icbox cbox_accl;

      if (accl > 1.2) throttle = 0;
      else {
            if (accl < 0.8) throttle = 8;
            else throttle = 20 * (1.2 - accl);
            }
      }

/* this maintains speed by the formula given in appendix   and set throttle */

void Throttle::maint_speed (int speed)
{
      int diff = desired_speed - speed;

      if(diff > 2) throttle = 0;
      else {
            if (diff < -2) throttle = 8;
            else throttle = 2*(diff +2);
            }
            printf("Throttle value %d\n",throttle);
      }

/* this correspondes to select speed function shown in fig */

void Throttle::select_speed()
{
      int speed;
      Icbox cbox_speed;
```

```
        *mbox_motion <= Mail(&Motion::reply_speed, cbox_speed) << 1;
        ~cbox_speed >> speed;

        desired_speed = speed;

        maint_speed(speed);
        }

/* this is invoked to activate the system,it calls select speed to select
current speed    after this throttle becomes an object    of type throt_crus, at
that time it only recvs the messages of type receivable    in that state. */

void Throttle::activate()
{
        select_speed();

        become (new throt_crus(5,mbox_motion, desired_speed, throttle));

        Reply(1);
}

void throt_idle::update()
{
printf("ERROR : update should not be called\n");
}

/* this is one of the valid transition in this state. please see the Figure */

void throt_idle::start_accl()
{
        *mbox_motion <= Mail(&Motion::update_throt) << ACCL;
        become (new throt_accl(5,mbox_motion, desired_speed, throttle));
        }

void throt_idle::resume()
{

        *mbox_motion <= Mail(&Motion::update_throt) << CRUS;
        become (new throt_crus(5, mbox_motion, desired_speed, throttle));

        }

/* following functions belon to class throt_idle, this is a subclass of
throt_idle    and calls the functions of that class like maint_accl, user in
this state is    accelrating */

void throt_accl::update()
```

```
{
        int accl;

        Msg >> accl;
        maint_accl(accl);
        }

void throt_accl::stop_accl()
{

        int speed;
        *mbox_motion <= Mail(&Motion::update_throt) << CRUS;
        select_speed();
        become (new throt_crus(5, mbox_motion, desired_speed, throttle));
        }

void throt_accl::braking()
{
        *mbox_motion <= Mail(&Motion::update_throt) << IDLE;
        become (new throt_idle(5, mbox_motion, desired_speed, throttle));

        }

void throt_accl::top_gear()
{

        *mbox_motion <= Mail(&Motion::update_throt) << IDLE;
        become (new throt_idle(5, mbox_motion, desired_speed, throttle));
        }


/* following functions belon to class throt_crus, this is a subclass of
throt_idle    and calls the functions of that class like maint_speed, user in
this state is    cruising at const speed */

void throt_crus::braking()
{

        *mbox_motion <= Mail(&Motion::update_throt) << IDLE;

        become (new throt_idle(5, mbox_motion, desired_speed, throttle));
        }

void throt_crus::top_gear()
{

        *mbox_motion <= Mail(&Motion::update_throt) << IDLE;
```

```
    become (new throt_idle(5, mbox_motion, desired_speed, throttle));
    }

void throt_crus::start_accl()
{
    *mbox_motion <= Mail(&Motion::update_throt) << ACCL;
    become (new throt_accl(5, mbox_motion, desired_speed, throttle));
    }


void throt_crus::update()
{

    int speed;

    Msg >> speed;
    maint_speed(speed);
    Reply(1);
    }
```

```
/*************************************************************************/
/*
/*    Class Name          -        task_activate
/*
/*    File Names -        Activate.c, Activate.h
/*
/*    PROGRAMMER    -        SANJAY KOHLI
/*
/*    DESCRIPTION     -        This class generates the receptionist for
/*                             the task Activate. It is invoked by
/*                             input handler.
/*
/*    DATE      -        Sept 25, 1989.
/*
/*************************************************************************/



/* activate.h */

class task_activate : public Iactor {
      Imbox* mbox_throt;
public:
      void recep();
      void recep1();
      void update_mbox();
};



/* activate.c */


/* this file contains the two receptionsits which are called under different
   circumstances, the first one recep is called to start the task in simulated
   system while recep1 is called during statistics estimates */


/* This function is called to update the address
   of throttle mailbox which comes in a message, to which the first message
   is sent by receptionist. */


void task_activate::update_mbox()
{
```

```
        int mbox; // to recv the mailbox in message

        Msg >> mbox;
        mbox_throt = (Imbox*)mbox; // Actually we got mailbox adress not int
        Reply(1);

        }

/* This receptionist is invoked by User_handler and start the execution of task
   activate by sending a message to throttle */

void task_activate::recep()
{

        Icbox cbox;

        *mbox_throt <= Mail(&Throttle::activate,cbox) ;
       //Invokethrottle_activate

        ~cbox;
        Reply(1);
        }

/* This receptionsit is called when the system is checking the execution time
   and memory requirement of the system. It will recv the throttle's mailbox
   id and sends a message to it and waits for the reply */

void task_activate::recep1()
{

        Icbox cbox;
        int mbox; // to recv the mailbox in message

        Msg >> mbox;
        mbox_throt = (Imbox*)mbox; // Actually we got mailbox adress not int

        *mbox_throt <= Mail(&Throttle::activate,cbox) ;
        // Invoke throttle_activate

        ~cbox;
        Reply(1);
        }
```

```
/*************************************************************************/
/*
/*    File Names -          rot.h, rot.c
/*
/*    PROGRAMMER    -       SANJAY KOHLI
/*
/*    DESCRIPTION   -       This class generates the receptionist for
/*                          the task shaft_rotation. It is invoked by
/*                          input handler.
/*
/*    DATE          -     Sept 25, 1989.
/*
/*************************************************************************/

/* activate.h */

class task_rot : public Iactor {
      Imbox* mbox_shaft;
public:
      void recep();
      void recep1();
      void update_mbox();
};


/* rot.c */

/* this file contains the two receptionsits which are called under different
   circumstances, the first one recep is called to start the task in simulated
   system while recep1 is called during statistics estimates */

void task_rot::update_mbox()
{

      int mbox;

      Msg >> mbox;
      mbox_shaft = (Imbox*)mbox;
      Reply(1);

      }

/* This receptionist is invoked by User_handler and start the execution of task
   activate by sending a message to throttle */

void task_rot::recep()
```

```
{

        Icbox cbox;

        *mbox_shaft <= Mail(&Shaft::update,cbox) ; // this task start with this

        ~cbox;

        Reply(1);
        }

/* This receptionsit is called when the system is checking the execution time
   and memory requirement of the system. It will recv the throttle's mailbox
   id and sends a message to it and waits for the reply */

void task_rot::recep1()
{

        Icbox cbox;
        int mbox;

        Msg >> mbox;
        mbox_shaft = (Imbox*)mbox;

        *mbox_shaft <= Mail(&Shaft::update,cbox) ; // this task start with this

        ~cbox;

        Reply(1);
        }

.fo on
:h1.Instrumented Actor Code
Following code contains file def.h and other classes as defined. The code for
def.h is given then code for the later classes is given.
.fo off

/* def.h */
/* This file contains only the defintions reqd to build the system */

extern int rand();
extern void srand(int);
extern long time();

/* set the the timing values and memory values for various ACT++ primitives */

#define SEND_TIMER 10
```

```
#define RECV_TIMER 10
#define NEW_TIMER  10
#define BECOME_TIMER 10


#define MBOX_MEM   sizeof(Mbox_)
#define CBOX_MEM   sizeof(Cbox_)
#define OBJ_MEM    1024 // stack size at present

#define MAX_FUN 10

#define Inew(actr,time) (timer.update_timer( NEW_TIMER), \
                timer.update_mem(MBOX_MEM + CBOX_MEM + sizeof(actr)), \
                *(new Imbox ((Actor_*)(new actr(time)))))

#define Rnew(actr,time) *(new Mbox_ ((Actor_*) (new actr(time))))

typedef void (*PFV)();

int randint(int u){
      int r = rand();
      if (r < 0) r = -r;
      return 1 + r%u;
      }
```

```
/*************************************************************************/
/*
/*    Class Name          -         Cond
/*
/*    File Name  -        cond.h, cond.c
/*
/*    PROGRAMMER    -         SANJAY KOHLI
/*
/*    DESCRIPTION     -         This class is responsible for handling branchinh
/*                              conditions, if and while. Cond is type of
/*                              object created. When object is created depending
/*                              on the type of input object of that type is
/*                              created.
/*
/*    DATE       -         Sept 25, 1989.
/*
/*************************************************************************/


/* Cond.h */

#define PROB 1
#define LOOP 2

class Cond {

      int loop;       // to keep when it is type loop
      float prob;     // to keep probability
      int type;       // type = PROB or LOOP
 public:
      Cond(int low, int high) {
            if(high > low)  loop = low + randint( high - low) ;
            else loop = 0;
            type = LOOP;}
      Cond(float p1)  { prob = p1; type = PROB;}
      int what() { return loop;}
      int operator!();
};

/* cond.c */

/* this function returns the random value between 0 and 1, it calls rand which
returns integer random numbers */
float drand(){
      float r1,r2;
      float f1;
```

```
    r1 = rand(); if(r1 < 0) r1 = -r1;
    r2 = rand(); if(r2 < 0) r2 = -r2;

    if (r1 < r2) f1 = r1/r2;
    else  f1 = r2/r1;
    return f1;

    }

/* operator ! returns true or false depending on probablity value > specified or
for    loop case whether loop is still > 0 or not */

int Cond::operator!(){

    if( type == PROB){
        if(drand() < prob)  return(1) ; else return(0);}
    else{
        if(loop-- < 1)  return(0) ; else return(1);}

    }
```

```
/*************************************************************************/
/*
/*    Class Name          -       Stat
/*
/*    File Name  -        info.h
/*
/*    PROGRAMMER    -     SANJAY KOHLI
/*
/*    DESCRIPTION     -   This class is responsible for collecting all the
/*                        statistics of execution time or memory requirement.
/*                        The object timer of type Stat is created in the
/*                        begining and it is updated when any primitive is
/*                        called.
/*
/*    DATE      -         Sept 25, 1989.
/*
/*************************************************************************/


class Stat {
      int time;
      int memory;
      int n;
      /* folowing params are usefule if running task several times to get
stats */

      // tot for all the runs
      int tot_mem; // tot_mem for all the runs
      int tot_time;

      // worst_case for all the tasks
      int worst_mem;
      int worst_time;
public:
      Stat() {time = memory = n = tot_time = tot_mem = worst_mem = worst_time
            = 0;}
      ~Stat() { }

      void reset() { time = memory = 0; }
      int what_time() {return time;}
      int what_memory() {return memory;}

      void update_mem( int val) { memory += val;}
      void update_timer(int val) { time += val;}
      void block(int val) { time +=  val;}
```

```
    void collect_stat() { tot_mem += memory; tot_time += time; n +=1;
                    if (time > worst_time) worst_time = time;
                    if (memory > worst_mem) worst_mem = memory;
                     }

    void print_stat() {
    printf("This case execution time %d and memory requirement  %d\n",time,
          memory);

    }

};
```

```
/*************************************************************************/
/*
/*    Class Name          -       Clock
/*
/*    File Name  -        clock.h, clock.c
/*
/*    PROGRAMMER    -     SANJAY KOHLI
/*
/*    DESCRIPTION   -     This class is responsible for creating clock
/*                        object. This object is responsible for invoking
/*                        periodic tasks. It keeps the mailbox addresses
/*                        and receptionsist functions with it.
/*
/*    DATE      -         Sept 25, 1989.
/*
/*************************************************************************/


// info.h

class Inst {

      int i; // index into tables
      int period[MAX_FUN]; // periods of the tasks
      int remainder[MAX_FUN]; // remainder
      float fraction; // the fraction of the load already added ie sum(c/p)
      void (*fun1[MAX_FUN])(); // address of function which creates the recep
      Mbox_* mbox[MAX_FUN]; // mbox address of reecp
public:
      Inst() {
            i = fraction = 0;
            remainder[1] = remainder[2] = 0; // should be more for more fns
            }
      ~Inst() { }
      void block(int val);
      void add_task( int, int, PFV, Mbox);
      void add_slot(int, int);
      // void add_task( int, int, PFV);
};


/* inst.c */

/* this function adds the task to the clock, so that they can be invoked
periodically    its variable are: time - is the period, ctime - execution time,
```

```
fun_point1 - is pointer to function, and mb is address of mailbox */


    void Inst::add_task(int time, int ctime, PFV fun_point1, Mbox_& mb){

        if((fraction + ctime/time) < 1){
            fraction = fraction + ctime/time;
            period[i] = time;
            fun1[i] = fun_point1;
            mbox[i] = &mb;
            i += 1;
            }
        else {  printf("TASK cannot be added\n");
            }
    }


    /* this function invokes the periodic tasks if their periods have come,
it does it is remebered in remainder of each task. when remainder >
period it invokes the tasks */

    void Inst::block(int val) {


        int j, rem;

        for(j=0;j<i;j++){

        remainder[j] += val; // calculate remainder for each task
        // if remainder greater then period so time to invoke the task
        if(remainder[j] >= period[j]){
            remainder[j] = 0;
            *mbox[j] <= Mail(fun1[j]) << randint(10);
            }
        }
    }

/* this adds the slot for the cases of aperiodic task, it just checks to see
    if the load is still less then 1 */


    void Inst::add_slot(int time, int ctime) {

        if((fraction + ctime/time) < 1){
            fraction = fraction + ctime/time;
            }
        else {  printf("Slot cannot be added\n");
```

```
        }
}
```

```
/*************************************************************************/
/*
/*    Class Name          -        Imbox
/*
/*    File Names -        Imbox.h,Imbox.c
/*
/*    PROGRAMMER    -        SANJAY KOHLI
/*
/*    DESCRIPTION     -        This class is defintion of Imbox, which is the
/*                             subclass of Mbox. It contains the primitives for
/*                             operator <=, and construction of Imbox.
/*
/*    DATE       -        Sept 25, 1989.
/*
/*************************************************************************/

/* #include "imbox.h" */

class Imbox : public Mbox_ {
friend Mail_;
friend Actor_;

public:
  Imbox(Actor_*);
  ~Imbox() { }
  void operator<=(Mail_&);
  };

/* imbox.c */

Imbox::Imbox(Actor_* act) : (act)
  {

  }
void
Imbox::operator<=(Mail_& mess)
{
 timer.update_timer( SEND_TIMER);
 Mbox_::operator<=(mess);
}
```

```
/*************************************************************************/
/*
/*    Class Name          -       Icbox
/*
/*    File Names -        Icbox.h, icbox.c
/*
/*    PROGRAMMER    -     SANJAY KOHLI
/*
/*    DESCRIPTION   -     This contains the definition of class Imbox.
/*                        it contains the primitives for all the Cbox primitives
/*                        A Icbox primitive updates the timer and calls
/*                        Cbox primitive.
/*
/*    DATE        -       Sept 25, 1989.
/*
/*************************************************************************/

/* "Icbox.h" */

/* These functions are all redfinitions of the functions in Cbox and they just
   call update_timer before calling respective functions of Cbox */

class Icbox : public Cbox {
public:

  Icbox() { timer.update_mem( CBOX_MEM); }

  Mail_& in();
  Mail_& in(Mbox_* from);
  Mail_& operator~()
  {
   timer.update_timer( RECV_TIMER);
   //printf("Updating ICBOX timer\n");
   return Cbox_::in();
  }
  void send_reply(Mail_& m)
  {

   timer.update_timer( SEND_TIMER);
   //printf("Updating ICBOX timer\n");
   Cbox_::send_reply(m);
  }
  void send_reply(int v)
  {
   timer.update_timer( SEND_TIMER);
   //printf("Updating ICBOX timer\n");
```

```
    Cbox_::send_reply(v);
  }
  void send_reply(double v)
  {
    timer.update_timer( SEND_TIMER);
    //printf("Updating ICBOX timer\n");
  Cbox_::send_reply(v);
  }

  void send_reply(char* v)
  {
    timer.update_timer( SEND_TIMER);
    //printf("Updating ICBOX timer\n");
    Cbox_::send_reply(v);
  }


  void send_reply(Actor_& v)
  {
    timer.update_timer( SEND_TIMER);
    //printf("Updating ICBOX timer\n");
    Cbox_::send_reply(v);
  }


  void send_reply(Mbox_& v)
  {
    timer.update_timer( SEND_TIMER);
    //printf("Updating ICBOX timer\n");
    Cbox_::send_reply(v);
  }

};

/* Icbox.c */

Mail_&
Icbox::in()
{
      timer.update_timer(RECV_TIMER);
      printf("Updating ICBOX timer\n");
      return  Cbox_::in();
}

Mail_&
Icbox::in(Mbox_* from)
{
```

```
    timer.update_timer(RECV_TIMER);
    printf("Updating ICBOX timer\n");
    return Cbox_::in(from);
}
```