# Methodology For Automating Distributed Hard Real-Time System Development

by

Daniel L.W. Liew

B.Sc., National University of Singapore, Singapore, 1981

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the Department
of
Computer Science



We accept this thesis as conforming
to the required standard

[redacted]

Dr. Gholamali C. Shoja, Supervisor

[redacted]

Dr. Micaela Serra, Departmental Member

[redacted]

Dr. Warren D. Little, Outside Member

[redacted]

Dr. Kin F. Li, External Examiner

Supervisor: Dr. Gholamali C. Shoja

# Abstract

Software designed to function in a hard real-time environment requires strict timing and precedence constraints. Software that is logically correct, i.e., implements the intended algorithm, may not function correctly if certain assumptions about timing characteristics are not met at run-time. A distributed hard real-time system provides reliability, availability and fault tolerance. However, the software for such systems is more complex in that it must also take into consideration resource contention as well as the associated communication delays.

This thesis presents a methodology for automating the development of distributed hard real-time systems. Heuristic algorithms for realizing the proposed methodology are introduced and implemented. Our strategy differs from previous work in that we also include provisions for deadline scheduling of messages, instead of assuming an unlimited communication bandwidth. We pre-allocate processor time slots for tasks as well as communication time slots for messages.

Actual examples of applying the implemented algorithm to pre-allocate tasks in a distributed environment such that timing constraints are met are presented and compared with previous work in this field. Areas for further research are also suggested.

Examiners:

Dr. Gholamali C. Shoja, Supervisor

Dr. Micaela Serra, Departmental Member

Dr. Warren D. Little, Outside Member

Dr. Kin F. Li, External Examiner

# Contents

**Abstract** ii

**Contents** iv

**List of Tables** ix

**List of Figures** x

**Acknowledgements** xii

**1 Introduction** 1

1.1 Problem description . . . . . 2

1.2 Objectives of the thesis . . . . . 4

1.3 Organization of the thesis . . . . . 5

**2 Background** 6

2.1 Introduction . . . . . 6

2.1.1 Virtual processor . . . . . 7

2.1.2 Processor sharing . . . . . 9
2.2 Related works . . . . . 10
2.2.1 Specification model for hard real-time systems . . . . . 10
2.2.2 Verification of real-time specification . . . . . 11
2.2.2.1 Leinbaugh's approach . . . . . 11
2.2.2.2 Ramamritham's approach . . . . . 12
2.2.3 Communication media . . . . . 13
2.3 Desired properties . . . . . 14
2.3.1 Modularity . . . . . 15
2.3.2 Reconfigurability . . . . . 15
2.3.3 Timing constraints . . . . . 16
2.3.4 Fault-tolerance . . . . . 16
2.4 Design methodology . . . . . 18

**3 Proposed Methodology** **20**

3.1 Introduction . . . . . 20
3.2 Specification model . . . . . 21
3.2.1 A graph based model . . . . . 22
3.2.1.1 Timing constraints . . . . . 22
3.2.1.2 Pipelined-ordered . . . . . 23
3.2.1.3 Process . . . . . 25
3.2.2 Blocking and non-blocking . . . . . 25

3.2.2.1 Non-blocking configuration . . . . . 26
3.2.2.2 Blocking configuration . . . . . 27
3.3 Process synchronization . . . . . 28
3.3.1 Monitor . . . . . 28
3.3.2 Process communication . . . . . 28
3.3.2.1 Conditions for rendezvous . . . . . 29
3.3.2.2 Communication primitives . . . . . 30
3.3.2.3 Proposed rendezvous mechanism . . . . . 30
3.4 The real-time scheduling problem . . . . . 33
3.4.1 The static schedule . . . . . 34
3.4.2 Scheduling of periodic tasks . . . . . 35
3.4.3 Scheduling of sporadic tasks . . . . . 36
3.4.4 Scheduling of monitor . . . . . 39
3.5 Run-time environment . . . . . 39
3.5.1 Run-time schedule . . . . . 39
3.5.1.1 Consideration for periodic tasks . . . . . 40
3.5.2 Run-time access of communication channel . . . . . 40
3.5.3 Modified VT/CSMA protocol . . . . . 42
3.5.3.1 Message transformation . . . . . 42
3.5.3.2 Outline of modified VT/CSMA protocol . . . . . 42

**4 The Task Allocation System** **45**

4.1 System structure for task allocation . . . . . 46

4.1.1 Feasible static schedule . . . . . 47

4.2 Task preprocessor module . . . . . 49

4.2.1 Worst-case computation time . . . . . 49

4.2.1.1 Restrictions for analyzable programs . . . . . 50

4.2.1.2 Calculation of 'worst-case' computation time . . . . . 50

4.2.1.3 Bounded loops . . . . . 52

4.2.1.4 The $\alpha$ function . . . . . 52

4.2.2 Task activation paths . . . . . 53

4.2.2.1 Task activation list . . . . . 53

4.2.2.2 Task table . . . . . 53

4.2.3 Sequencing of tasks . . . . . 55

4.2.4 Computation of pseudo window . . . . . 57

4.2.4.1 Algorithm . . . . . 57

4.2.5 Pseudo windows with no communication cost . . . . . 58

4.2.6 Computation of task priority . . . . . 60

4.3 Task allocation module . . . . . 60

4.3.1 The basic strategy . . . . . 62

4.3.2 Computing of temporary pseudo window . . . . . 63

4.3.2.1 The enable time . . . . . 64

4.3.2.2 Temporary pseudo ready time . . . . . 66
4.3.2.3 Selection of task . . . . . 66
4.3.3 Task clustering guidelines . . . . . 67
4.4 Resource allocation module . . . . . 68
4.4.1 Communication time-slot sub-module . . . . . 68
4.4.1.1 The communication time-slot algorithm . . . . . 68
4.4.2 Mutual exclusion sub-module . . . . . 70
4.4.2.1 Algorithm for mutual exclusion . . . . . 70

**5 Results and Analysis** **75**

5.1 Processing the specification file . . . . . 76
5.2 Initial results . . . . . 79
5.2.1 Earliest pseudo ready time . . . . . 79
5.2.1.1 Task priority . . . . . 80
5.3 Allocation of tasks . . . . . 81
5.3.1 Execution windows . . . . . 81
5.3.2 Temporary pseudo windows . . . . . 82
5.3.2.1 Available task set . . . . . 82
5.3.2.2 Binding predecessor task . . . . . 82
5.3.2.3 Temporary pseudo ready . . . . . 84
5.3.2.4 The latest enable time . . . . . 84
5.3.3 Selection of task . . . . . 86

5.3.3.1 Pseudo deadline . . . . . 87
5.3.4 Allocation of communication time-slots . . . . . 88
5.3.4.1 Computation of communication time-slot . . . . . 90
5.3.5 Final result . . . . . 90
5.4 Comparison of task allocation heuristics . . . . . 91
5.4.1 Cheng's heuristic algorithm . . . . . 91
5.4.2 Assignment of tasks . . . . . 91
5.4.3 Assignment of communication time-slots . . . . . 93
5.4.4 Final analysis . . . . . 95

**6 Conclusion** **97**

6.1 Summary . . . . . 97
6.2 Current status . . . . . 98
6.3 Future research . . . . . 99

**A Decomposition Module** **107**

A.1 Approach . . . . . 107
A.1.1 Specification file . . . . . 108
A.2 Decomposition strategy . . . . . 108
A.2.1 Example . . . . . 110

**B Pseudo Code and Examples** **116**

**C Glossary** **121**

# List of Tables


4.1 Formulas for the calculation of 'worst-case' computation time 51

5.1 The task priority table . . . . . 80

5.2 Results of temporary pseudo windows . . . . . 83

5.3 Assignment of execution windows . . . . . 88

5.4 Timing constraints of external messages . . . . . 89

5.5 Assignment of communication time-slots . . . . . 89

5.6 Results generated by Cheng's heuristic algorithm . . . . . 92

5.7 Timing constraints of external messages . . . . . 94

5.8 Failure in the asssignment of time-slot . . . . . 95

A.1 Primitives and functions used in the process and monitor constructions . . . . . 115

C.1 Abbreviations used in the task allocation heuristics . . . . . 122

C.2 Abbreviations used in the time-slot allocation heuristics . . . 123

C.3 Abbreviations used in the mutual exclusion algorithm . . . . 124

C.4 Functions used in all the heuristic algorithms . . . . . 125

# List of Figures


3.1 An example of a precedence task graph . . . . . 24

3.2 An example of a non-blocking configuration . . . . . 26

3.3 An example of a blocking configuration . . . . . 27

3.4 An 'equivalent' task for a sporadic task . . . . . 38

4.1 The structure of the task allocation system . . . . . 48

4.2 An example of a task activation list . . . . . 54

4.3 An example of a task table . . . . . 56

4.4 The derivation of the pseudo ready time . . . . . 59

4.5 The derivation of the pseudo deadline . . . . . 59

4.6 The computation of the task priority . . . . . 61

4.7 Assignment of communication time-slots . . . . . 71

4.8 The derivation of the reserved slot . . . . . 74

5.1 A run-time execution of the task allocation system . . . . . 77

5.2 The assignment of tasks using Cheng's heuristic algorithm . 93

5.3 The assignment of tasks and messages for the two heuristics 96

A.1 Format for the specification file . . . . . 111

A.2 Examples of specification files . . . . . 112

A.3 Processes for the three input nodes . . . . . 113

A.4 Monitor and process for the common node . . . . . 114

B.1 Pseudo code for the task allocation heuristics . . . . . 117

B.2 Outline of the communication time-slot assignment algorithm 118

B.3 Pseudo code for the subroutine time_slot_algo() . . . . . 119

B.4 Example given in Mok's paper[Mok84b] . . . . . 120

B.5 Assignment of tasks and messages for Mok's example . . . . . 120

# Acknowledgements

I wish to thank my thesis supervisor Dr. Gholamali Shoja for his unwaving support and encouragement towards me over these two years. His patience and confidence in my ability are perhaps the best support that a graduate student can hope for. I shall always remember his advise: "There is no magic, but hard work". Without his constant urgings and helpful suggestions, this thesis would never be completed.

And of course, without the love, the joy and the motivation of my wife Siu Guan, daughter Dawn and my new born son Jason, I might never have graduated. Their faith in my ability and the thought that they are always behind me is sometimes the only thing that keeps me going.

Financial support from Ngee Ann Polytechnic in the form of postgraduate scholarship made this work possible. And above all, I want to thank God for His providence and goodness towards me.

# Chapter 1

# Introduction

We begin by defining the terms *hard real-time* and *distributed systems*, before describing the problem we have tried to address in this thesis.

The term *hard real-time* is used to describe systems that must meet strict timing demands. Such systems are required to respond to a variety of non-deterministic external events, within specified real-time limits[Faulk88].

Hard real-time systems are used in applications such as production control, air traffic control, robotics, tele-communication, medical and patient monitoring, and embedded tactical systems for military and space applications. Hard real-time systems differ from soft real-time systems in that the consequences of missing a deadline in a hard real-time system is generally disastrous. For example, the plane may crash or a plant may explode if certain events are not responded to within a time limit. In soft real-time systems on the other hand, untimely delivery of results may merely cause inconvenience to the user. A good example of a soft real-time system is the Airline Reservation System, where the client or the travel agent may get annoyed for waiting few minutes to find out about the available of seats on a flight, but this

causes no disaster.

Definition of a *distributed system* however, is not as precise as that of a hard real-time system. Our definition of a distributed system is similar to that put forward by Le Lann[Davies83a]. He defines a distributed system as one comprised of multiple processing elements, interconnected, without a hierarchical control structure, and existence of processes with disjoint address spaces communicating via explicit message passing. Extensibility, better availability and reliability, resource sharing, and potential for fault-tolerance are among the advantages of a distributed system.

## 1.1 Problem description

Existing approaches to designing systems for hard real-time environments are rather ad hoc. There are few tools to verify that the specified timing constraints can invariantly be satisfied at run-time. Performance evaluation is accomplished either by stochastic simulation or by actual measurement on prototype testbeds [Chu80]. System performance is often improved by fine tuning certain system parameters. When specifications cannot be met, structural modifications may become necessary so that at least the major performance objectives can be achieved. While this iterative approach may suffice for building non critical data processing systems, it may not be suitable for systems which must function in a hard real-time environment. Unless the interactions among different components of a system are well understood and taken into consideration during design process, there is no easy way to satisfy multiple stringent timing constraints by fine tuning the system after it has been designed.

The additional dilemma of such ad hoc methods of development is the problem of software maintenance. Every update to the system may result in changes to almost all components of the real-time software. Translation to a process model from a formal

specification is desirable because the process model is well understood. However, there is no guarantee that the resulting process model will produce code that will meet the timing constraints at run-time.

Since most safety-critical systems are hard real-time, then in addition to high performance, reliability and fault-tolerance are also of prime importance in such systems. It is therefore logical that we should look at distributed systems as potential candidates for implementing reliable hard real-time systems.

Distributed *real-time* systems introduce additional constraints into the already complex problem of sharing work over a set of distributed processors. First, when a group of interacting tasks are involved, end-to-end timing requirements must include the precedence constraints on top of the computational requirements. Second, tasks may fail to complete in the required time because of delays while awaiting for shared resources. In distributed real-time systems, contention for communication channels is a particularly critical problem. Furthermore, communication channels are often subject to errors, which often trigger additional and unexpected timing delays at the application level.

Third, many task scheduling schemes are not suitable for the distributed hard real-time environment. For example, many schedulers are designed to prevent resource starvation and deadlock, but do not guarantee that tasks meet deadlines [Lomet80]. Some do not always correctly manage task priorities to ensure that high priority tasks complete within specified deadlines [Locke88]. Often they do not deal with priority inversion, a situation in which agents for low priority tasks are scheduled ahead of high priority time critical tasks. A glaring example of priority inversion is the task definitions of Ada [Ada80]. Ada mandates the strict observation of priority in the allocation of hardware resources and ignores the priority assignment for logical resources. It ignores the priority on entry queues and on the selection calls. In

fact, following Ada's tasking model, deadlines would be missed at a very low level utilization of 25% [Locke88].

This thesis tries to introduce methods which will make the development of distributed hard real-time system more manageable and moves some steps further than the work already accomplished in this field.

## 1.2 Objectives of the thesis

The main objectives of this thesis are to propose a methodology for automating the development of distributed hard real-time systems and to provide heuristic algorithms and tools necessary for applying such a methodology. More specifically our goals are:

1. To propose a methodology that includes:
    - the ability to specify any hard real-time system precisely according to a given performance specifications.
    - a method to translate the hard real-time specifications into a process model consisting of a group of tasks with precedence constraints.
    - an approach for scheduling periodic and asynchronous tasks with precedence and timing constraints.
    - a facility for concurrency control and interprocess coordination which would support control structures commonly required for hard real-time systems.
    - a suggestion for the run-time environment for using the methodology.
2. To develop appropriate heuristics to allocate available processor times and communication bandwidths. The heuristics should provide the ability to determine

whether a feasible schedule for tasks and messages exists. Such a schedule must take into consideration the following requirements.

- The deadlines of periodic tasks and the response times of sporadic tasks must both be met.
- The precedence of tasks must be kept.
- The allocation of communication media for external messages must be provided for.
- The problem of mutual exclusions must be considered.

## 1.3 Organization of the thesis

The remainder of this thesis is organized as follows. Chapter 2 provides a review of previous work in this field. In Chapter 3, our proposed methodology for specifying and formulating hard real-time systems is given. Chapter 4 presents the structure of our task allocation system and gives a detail discussion of the approach as well as the heuristics and algorithms used in the system. Chapter 5 gives the results generated by the task allocation system and makes a comparative study of our task allocation heuristics with a recent method proposed by Cheng [Cheng86]. Finally, Chapter 6 concludes the thesis by providing a summary of the achievements of this thesis and directions for further research in this field.

# Chapter 2

# Background

## 2.1 Introduction

In recent years, a lot of research has been directed at solving the problem of producing software for a hard real-time environment [Lein82][Mok83][Faulk88]. Most of this research falls into one or more of the following three categories: specification techniques, language concepts, design disciplines and related scheduling techniques.

Specification techniques are concerned mostly with the functional completeness and consistency of the application domain descriptions of real-time systems. Examples are [Alford77],[Cohen78], [Hen80]. The specification of stringent timing constraints is treated in some detail in [Mok84a]. [Faulk88] describes a formal specification for the development of avionics software used in the Navy's A-7E aircraft including timing specifications. Some programming languages have been augmented by scheduling primitives to support real-time programming, eg., PL/1 [Barnes79], Concurrent C++ [Gehani88]. Others like [Dijks68], [Brinch78] and [Hoare78] have investigated special language concepts to represent concurrent processes and to facil-

itate real-time programming. However, few attempts have been made to provide a methodology for transforming a set of requirements specifications, including timing and communication constraints, into a viable assignment of tasks for a distributed system.

For our thesis purposes, the cited literature on real-time specifications fall short of defining a computation model with interprocess communication which can be decomposed mechanically for feasibility analysis in a distributed environment. Further improvement is needed to specify a computation model with timing and communication constraints. Although [Mok84b] addresses issues similar to our thesis topic, he does not include communication constraints and the analysis of their effects on the model. In particular, his work was centered upon the scheduling of real-time tasks for uniprocessors and multiprocessors with shared memory environment.

Works that are most closely related to our research are the design methodologies that have been proposed for hard real-time programs. Some of these methodologies have limited use for hard real-time applications in that they do not address the problem of verifying compliance to timing and communication constraints specifications. Almost all of these methods are process-based and can be roughly categorized as adopting one of two approaches namely, virtual processor and processor sharing [Davies83a].

### 2.1.1 Virtual processor

In this approach, each process is presumed to be running on a dedicated processor. The objective is to guarantee bounded completion time for all required computations. To this end, the designer needs to guarantee that no deadlock can result from the control structure of the program and that no part of the computation will be de-

nied progress indefinitely. The problem of verifying compliance to timing constraints specifications is deferred until run-time and must be solved for the actual scheduling strategy in an implementation, presumably by computing the worst case bounds for all completion times and comparing them against the specifications.

The virtual processor approach is viable only if the premise about resource availability is valid, and only up to the point that interprocessor communication does not become a bottleneck. Even if there are enough processors so that each process can be run on a dedicated processor, timing constraints may still be violated because of the communication delay between processors. For more demanding applications, the simplistic approach of assigning one process to one processor may not work. The basic problem of balancing computing and communication costs cannot always be ignored.

Some attempts have been made to formulate the processor allocation problem in the presence of stringent timing constraints, e.g., [Chu80], [Ma82]. However, none of the published formulations seem to be satisfactory since all of them are formulated with algebraic constraints on mean value parameters such as communication bandwidth and processor utilization factor. These parameters are in principle derivable from the process model, but unfortunately, they are more useful for average time rather than worst case analysis. The bursty nature of many real-time applications is such that there is no guarantee that individual timing constraint will be met even if the average load does not exceed either processor or communication capacity. In this thesis, we shall provide a methodology for the problem of task assignment that is more amendable to the requirement of the distributed hard real-time environment.

### 2.1.2 Processor sharing

In this approach, processes are expected to share resources subject to known scheduling disciplines and usage restrictions which are part of the design methodology. Since the scheduling disciplines are fixed, the run-time behavior of the system is predictable and this offers potential for exploitation. First, the designer need not guarantee that a system is inherently deadlock-free or does not have other undesirable properties; it is sufficient that the system will function properly under the given resource scheduling disciplines. For example, the system scheduler might require that all shared resources be explicitly declared and apply Lamport's algorithm for resource allocation [Lamport78]. Second, there may be asymmetry in the resource scheduling disciplines which can be exploited to favor processes with tighter timing constraints. This can be facilitated by introducing scheduling attributes, e.g., deadlines for interfacing with the system scheduler.

[Wei80] and [Lein82] provide some examples of the processor sharing approach. In [Wei80], processes are assigned deadlines and period attributes and are run on a single processor. The scheduling disciplines used are variations of the earliest deadline algorithm which always runs the ready process with the nearest deadline, or the rate monotonic static priority algorithm which assigns higher priorities to processes with higher repetition rates. In [Lein82], processes may also contain device segment (I/O delays) and resource segments (critical sections) which are scheduled on a first-come-first-served basis.

It should also be noted that if we can find an optimal scheduling algorithm for every resource, then there is no reason why the processor sharing approach should not be adopted with the optimal scheduling disciplines. However, it has been shown [Mok78] that no optimal scheduling algorithm exists for a multiprocessor, unless start times,

computational requirements, and deadlines of all processes are known in advance.

## 2.2 Related works

### 2.2.1 Specification model for hard real-time systems

[Mok84a] presents a graph based model of computation for the hard real-time environment. He uses an acyclic digraph to represent a timing constraint graph which defines the precedence relation of the computational events that must occur in order to satisfy a timing constraint. The nodes and edges of a timing constraint graph denote the execution times of the corresponding functional elements and the corresponding communication delays, respectively.

A periodic or sporadic task with given set of timing constraints may be mapped into a compatible communication graph. Sporadic tasks are converted into equivalent periodic tasks and monitors are created for each functional element that occurs in two or more timing constraints. This computational model is decomposed into equivalent process-based model and scheduling results are applied by mapping each timing constraints into the model [Mok84b].

One of the virtues of the graph based model is that all the data dependencies are made explicit. Mok's model is suitable for implementation by a set of concurrent processes running on one or more processors with common access to a shared memory [Mok83]. However, the scheduling of tasks does not take into consideration the blocking which takes place when tasks assigned to different nodes attempt to access a shared communication media such as the *Ethernet* [Metcalf76].

### 2.2.2 Verification of real-time specification

Whereas many real-time programming languages and systems offer run-time support for the verification of timing constraints, very few perform pre-run-time, compile-time or configuration-time verifications. [Lein82] and [Ramam84] present two different approaches for statically determining the real-time feasibility of programs.

#### 2.2.2.1 Leinbaugh's approach

Leinbaugh presents a distributed real-time programming model consisting of a set of tasks whose segments require processor time, critical section accesses and I/O device resources. Associated with each task is a deadline, specifying the time within which the task must complete once it has started, and a minimum period, which is the minimum allowable time between successive starts. A task is started upon receipt of an external signal or upon completion of other tasks. Each task is subdivided into a set of sequential and/or parallel segments, each requiring a specific CPU time and possible access to a particular critical section or any I/O devices. A segment that requires access to a critical section must obtain access to it before executing and only relinquishes it after completing. Run-time scheduling of tasks is driven by resource contention and by the software priorities that are associated with all tasks. Critical sections and I/O devices are assigned on a first-come-first-served basis. Each computer node in the distributed network is independently connected with each other. Inter-node messages are treated as external signals by the destination node [Lein82].

Leinbaugh describes how a pessimistic timing analysis can be performed on instances of this model. The analysis calculates an upper bound on the response time of each task by determining in the worst-case, the degree to which the activities of the other tasks can affect the task being examined. These upper bounds are compared

with the specific task deadlines, and all detected timing violations are recorded.

#### 2.2.2.2 Ramamritham's approach

[Ramam84] presents a somewhat simpler model. A program consists of a collection of periodic and sporadic tasks. A periodic task is executed at regular time intervals whereas a sporadic task executes exactly once and can be started at any time. Associated with each task is a deadline and an execution time. Periodic tasks are assigned to execute on specific processors whereas sporadic tasks can arrive at any processor, at any time. This model does not allow tasks to preempt each other, nor to compete for resources other than the CPU.

A static analysis is performed to determine if a feasible non-preemptive schedule (one in which all tasks always meet their deadlines) exists for the periodic tasks that are assigned to each processor. For a specific processor, the start time and deadline of every periodic task instance, occurring within a window of time equal to the least common multiple of the periods of all periodic tasks, is calculated. The analysis determines for each periodic task instance in the window, the latest possible time it can start execution, assuming that tasks are scheduled to execute just in time to meet their deadlines. If the latest execution time of a task is greater than or equal to its start time, then it is guaranteed to execute. A feasible schedule exists for each set of periodic tasks if all tasks instances in the analysis can be guaranteed. At run-time, guaranteed periodic tasks are scheduled to meet their deadlines according to a non-preemptive earliest-deadline scheduling policy.

The scheduling of sporadic tasks is handled at run-time. When a sporadic task arrives at a node, the run-time scheduler at that node attempts, by using an algorithm that is similar to the one used in the static analysis, to guarantee its execution. If

timely execution of the task cannot be guaranteed, then a bidding process is initiated to determine if timely execution of the task can be guaranteed on another node [Zhao85].

This static analysis has the property that if it can guarantee all periodic tasks executing on a node, then a non-preemptive earliest-deadline scheduling algorithm will, at run-time, schedule all periodic tasks within their deadlines.

### 2.2.3 Communication media

In distributed hard real-time systems, multiple access networks are commonly used. In this type of network, stations transmit messages via a shared channel. Only one message can be successfully transmitted over the channel at any time. A collision occurs if, at any time, two or more messages are transmitted on the channel. No message can be received correctly in the event of a collision.

Based on how the collisions are handled, multiple access communication protocols can be broadly divided into three categories [Tanen88]:

1. **Inference avoiding protocols:** These protocols operate without taking past history of the channel into account. This category includes ALOHA [Abrams70] and various CSMA protocols [Lam80][Molle85].

2. **Inference seeking protocols:** These protocols make inference on the collision history, and usually solve collisions by partitioning some parameter space of messages. Various tree, window, stack and urn protocol [Klein78] belong to this category.

3. **Deterministic or Collision-free protocols:** These protocols work in such a way that collisions do not occur at all. The Time Division Multiple-Access

Protocols (*TDMA*) [Cape79], the Bit-Map Protocol [Klein83] and the Broadcast Recognition and Alternative Priorities Protocol [Chlam79] are examples of protocols in this category.

The majority of communication protocols found in the above three categories do not directly address timing constraints. An example belonging to the third category is *MARS* [Kopetz85]. In the case of *MARS*, a *TDMA* media-access strategy is used to provide a deterministic, load-independent, and collision-free method for media access. Hard real-time tasks are scheduled with fixed period and it does not handle sporadic tasks. The periodic cycles of all tasks are synchronized in advance with the *TDMA-slots* to optimize the system-response behavior. However, the periodic cycles of the tasks must be a multiple power of two of the smallest possible periodic cycle. Furthermore, only one message can be sent per communication-slot of the media. It does not handle messages with variable length or arbitrary deadlines. Protocol for the hard real-time communication belonging to the first category has also been developed and reported in [Zhao86] and is discussed further in Section 3.5.

## 2.3 Desired properties

The design goal of a hard real-time system is not only high performance but determinism and predictability of the system behavior. This section identifies some properties that we feel are desirable in a distributed real-time environment. These properties, however, are not necessarily relevant for all real-time systems, but, rather, derive from the particular environment in which we are interested. Such an environment, exemplified by a multisensory factory automated system, comprises processors of varying sizes and capacities. Of more significance is the fact that such systems are highly likely to undergo changes and development updates. The system itself is likely

to be modified as devices and processors are upgraded. In addition, the system is required to function in a wide variety of applications requiring tailor made application programs. Consequently, changes rather than stability is a dominant characteristic.

### 2.3.1 Modularity

In a distributed real-time environment, several levels of modularity are desirable. First, a real-time program must often perform a number of relatively independent tasks. In order to simplify program development, it should be possible to program each of these tasks as separate modules. Specifically, a distributed real-time system should enable programs to be built from a set of independently designed modules. Each module can implement a specific function and should interact with other modules through a well-defined interface. Finally, the programming language that is used to program the various components of a real-time program, should have built into it modular features such as information hiding and support for data abstractions [Gehani88]. These features will enable the programmer to isolate implementation details such as machine dependencies from the higher-level parts of the program [Davies83b].

### 2.3.2 Reconfigurability

Reconfigurability refers to ease with which changes can be incorporated into a program. In order to keep program development and maintenance reasonably efficient and economical, a distributed hard real-time system must provide a high degree of reconfigurability. In particular, it should be possible to implement small behavioral and structural changes to a program without having to do extensive reprogramming and recompilation of program modules [Magee83].

### 2.3.3 Timing constraints

A distributed hard real-time system must supply the means of specifying, verifying and enforcing timing constraints [Mok88]. Timing constraints can be imposed on module completion and on module periodicity. The verification of timing constraints can be performed statically at configuration time, and dynamically at run-time. The enforcement of timing constraints at run-time requires first, the operating system or the real-time executive to have a scheduling policy that gives priority to the most urgent task in the program. Second, the communication primitives for synchronization of tasks must include length and deadline of messages. Third, the underlying mechanism for the shared access media should have a earliest-deadline or maximum-laxity protocol [Zhao86].

### 2.3.4 Fault-tolerance

In order to provide reliability despite the presence of faults, measures for fault-tolerance must be adopted in a distributed hard real-time system. The key to tolerating component failures is replication. The idea is to replicate each component to such a degree that the probability of all replicas failing becomes acceptably small. The advent of inexpensive distributed computing systems makes replication an attractive and practical means of tolerating hardware crashes.

The two architectures for fault-tolerant software are primary-standby systems and modular redundancy. In the primary-standby scheme, only a single component functions normally; the remaining replicas are on standby in case the primary fails. With modular redundancy, each component performs the same function; there is some form of voting to mask failures.

An example of the primary-standby architecture is the *ISIS* project developed

at Cornell [Birman84]. The *ISIS* uses primary-standby architecture to implement replicated objects. In each interaction with a replicated object in *ISIS*, one replica plays the role of coordinator, and only it performs the operation. The coordinator then uses a two-phase commit protocol to update the other replicas.

The mechanism used in primary-standby schemes, to allow a standby to take over after the primary crashes, are isomorphic to crash recovery mechanisms based on stable storage [Lampson83]. Under this isomorphism, a standby corresponds to a stable storage while the primary continues to function, but assumes the role of the recovering machine when the primary fails.

This scheme may not be suitable for distributed hard real-time system because of the additional overhead of checkpointing, crash recovery and communication delays in the event of failures. In the event of a failure, crash recovery and communication delays may result in missing deadlines. Furthermore, the various possibility of reconfiguration in the event of a processor crash may be difficult to predict and analyze during system configuration time.

A recent example of the modular redundancy scheme is the *Circus* project developed at Berkeley [Cooper85]. Cooper uses a set of replicas for a module called *troupe* and uses replicated procedure call mechanism to handle the many-to-many pattern of communication between troupes. In *Circus* system, crash recovery mechanisms are required only for total failures, in which every troupe members crashes. The probability of total failures can be made arbitrarily small by choosing an appropriate degree of replication.

The *Circus* scheme is suitable for implementing fault-tolerance in a distributed hard real-time system because first, the cost of replicated procedure call for communication between troupe is deterministic and can be computed at pre-run-time. Second, there is no additional time delays or overhead incurred in the event of failures since

clients receive notification of the crashes of troupe members and can proceed normally. Third, since *Circus* guarantees replication transparency (i.e., troupe members are required to behave deterministically), it does not have to worry about the side effects of replicas when analyzing the timing constraints of the program.

## 2.4 Design methodology

The methodology for designing a distributed hard real-time system can be tackled by many approaches. For example, one can select a programming language and provide tools for verifying a program against the timing constraint specifications when it is run on some target hardware configuration. Alternatively, one can dictate a set of software design rules and specify restrictions on the resource usage so that there is an efficient algorithm to determine, for any hardware configuration, if some program written to conform with the rules can meet the specified timing constraints [Sha88].

In general, a well defined methodology must have a model of computation in terms of which the computation requirements of the application domain can be expressed. The function of a methodology is to provide higher-order rules and algorithms for feasibility analysis and to suggest a solution when appropriate. For this purpose, the model of computation must be sufficiently precise so that software tools can be brought to determine the feasibility of the design. A simple measure for evaluating the effectiveness of a design methodology is its efficiency, i.e., the range of stringent design requirements that can be met by adopting the design methodology [Ramam87].

The effectiveness of a design methodology is indicated by the variety of system resources that can be adequately characterized, i.e., given the computational requirements expressed in an instance of the computation model, a decision procedure should exist which determines whether there is sufficient resource of every type to satisfy the

requirements. Ideally, the computation model should be able to characterize the demand load on any kind of existing or to-be-invented resource by a distributed hard real-time system. More realistically, we would like our design methodology to be able to integrate systems using conventional hardware, e.g., shared media access: *Ethernet* bus [Stan88]. We describe our proposed methodology in the next chapter.

# Chapter 3

# Proposed Methodology

## 3.1 Introduction

Our proposed methodology for specifying hard real-time systems is intended to achieve the following objectives:

- provide a specification model which can be used for verification of the logical correctness of a system.
- allow feasibility analysis of the timing, communications and resources constraints given in the specification.
- provide an integrated approach for scheduling processor, communication and other shared resources.

We shall discuss the proposed methodology in this chapter, the task allocation system in Chapter 4 and the results and analysis of the the task allocation system in Chapter 5.

Our proposed methodology includes:

1. use of a graph based model for the specification of a real-time system as described in [Mok84b]. However in our model, the communication path between tasks may either be blocking or non-blocking as described in the next section.

2. a process [1] model of distributed programs transformed from the graph based model [Mok84a].

3. a proposed model for process communication in distributed real-time environment.

4. a task-oriented scheduling model where the assignment of both periodic and sporadic tasks takes into consideration a shared access communication media and other shared resources.

5. a model for the run-time environment which supports minimum laxity or earliest deadline scheduling of both the tasks and the inter-task messages.

## 3.2 Specification model

For specification of a hard real-time system, we use a graph based model as described in [Mok84b]. The main advantages of Mok's model are:

1. The ability to verify logical correctness of a real-time system.

2. It provides a basis for decomposing a computational model into a process model.

[1] We distinguish between a process and a task as explained in Section 3.2.1.3

3. It can be used to formulate and study relevant problems in resource scheduling with critical timing constraints.

4. It provides a precise formulation of the processor allocation problem for hard real-time environment.

Our proposed model however has an additional capability of defining *blocking* and *non-blocking* tasks. This is useful in cases where the real-time system consists of both blocking and non-blocking rendezvous between tasks. Such information is also extremely useful for decomposing the computation model into a process model and in the tasks allocation problem. We shall discuss this in Section 3.2.2.

### 3.2.1 A graph based model

A graph based model $M$ is an ordered pair $(G, T)$ where $G$ is a communication graph and $T$ is a set of timing constraints. Specifically, $G = (N, E, W_v)$ is a digraph where $N$ and $E$ are respectively the node and edge sets and $W_v$ is a function which assigns a non-negative weight to each node in $v$. Also $T = T_p \cup T_a$ is a finite set of timing constraints each of which is a tuple $(C, p, d)$ where $C$ is a timing constraint graph, and the non-negative integers $p$ and $d$ are period and deadline of the timing constraint respectively.

#### 3.2.1.1 Timing constraints

A timing constraint graph is meant to define the precedence relation of the computational events that must occur in order to satisfy a timing constraint. The nodes and edges of a timing constraint graph denote the execution of the corresponding functional elements and transmission of data in the communication graph, respectively.

The computation time of a timing constraint $(C, p, d)$ is the sum of all the weights of the nodes in $C$.

If a timing constraint $(C, p, d) \in T_p$, then it is activated automatically every $p$ time units, starting from $time = 0$ (i.e., the timing constraint is periodic). If $(C, p, d) \in T_a$, then it can be activated at any integral time instant $t$ with the provision that two successive activations on the same timing constraint must be at least $p$ unit time apart (i.e., the timing constraint is asynchronous or sporadic). If timing constraint $(C, p, d)$ is activated at $time = t$, then the group of tasks represented by the timing constraint graph must be executed in the interval $[t, t + d]$.

Figure 3.1 shows an example of a precedence graph specified in a graph based model. In this example, the precedence graph contains eight tasks, $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $T_7$ and $T_8$. Task $T_1$ is the beginning task and has no predecessor task but has an external input edge $t_1$. Task $T_8$ is the ending task which has no succeeding task but has an external output edge $t_8$. In this sample group of tasks, each node denotes a task's worst case computation time and each edge denotes the worst-case communication time between a pair of communicating tasks. The deadline of the sample group is given as 50 which is the latest elapsed time for all the tasks in the group to be completed upon activation. This graph is created using a modified version of Mok's critical timing constraint method[Mok84b] of process decomposition as described in appendix A.

#### 3.2.1.2 Pipelined-ordered

We further assume that the real-time computations are pipeline-ordered in the sense that:

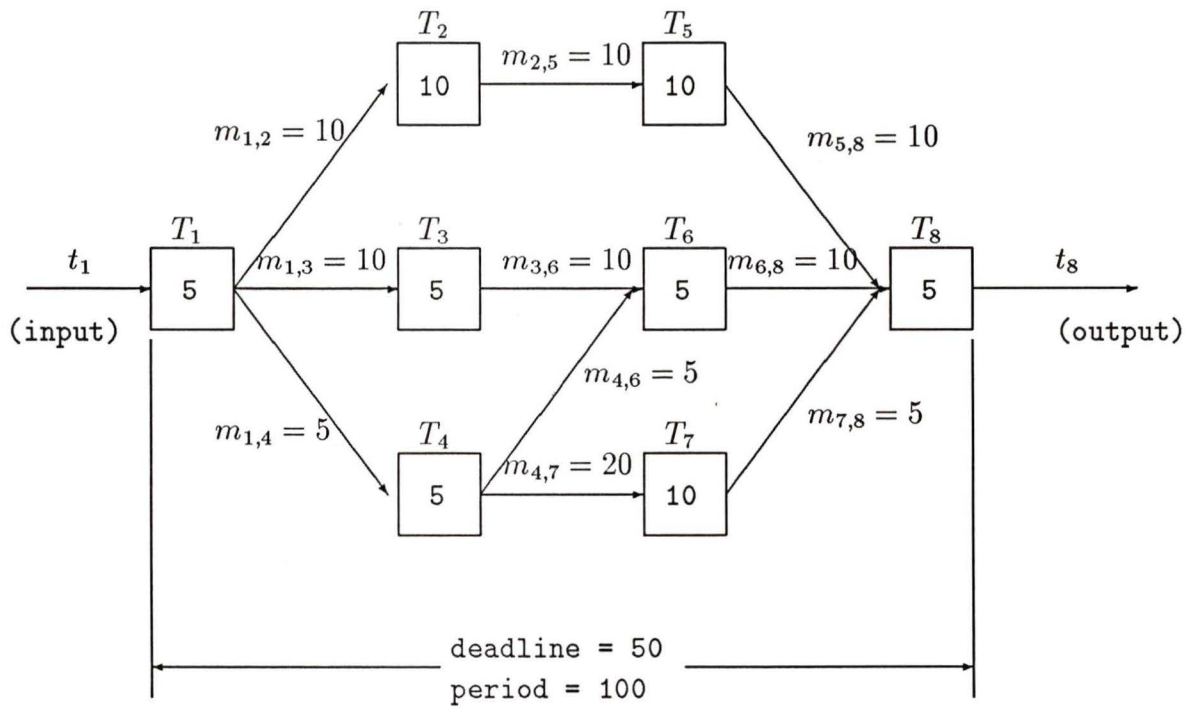



Figure 3.1: **An example of a precedence task graph**

1. Two executions of a functional element must have distinct start-times and that the execution which has an earlier start-time must finish earlier than the other.

2. Two data transmissions from a functional element $u$ to another functional element $v$ must be sent at distinct instants at the site $u$ and the earlier transmission must also be received earlier at the site of $v$. In order to enforce pipeline ordering, a monitor is created for each functional element that occurs in two or more timing constraints.

#### 3.2.1.3 Process

We use the term process to mean a task which is bracketed by communication primitives added by our pre-allocation system. Specifically, a task is transformed to a process when the header file of the task is updated with communication information supplied by our pre-run-time task allocation system. See Appendix A for examples.

Interprocess synchronization is achieved through communication primitives which may appear only between executable blocks of code. These communication primitives are used to pass information among processes or for coordination purposes. Their semantics is important only to the extent that they impose certain scheduling restrictions which will be defined in Section 3.3.2.

### 3.2.2 Blocking and non-blocking

In order to specify real-time systems with both blocking and non-blocking tasks, we allow two types of edges to be specified namely, *blocking or non-blocking.* In a blocking edge, the succeeding task must receive synchronization signals of completion from all precedence tasks. In the case of non-blocking edge, any predecessor task may activate

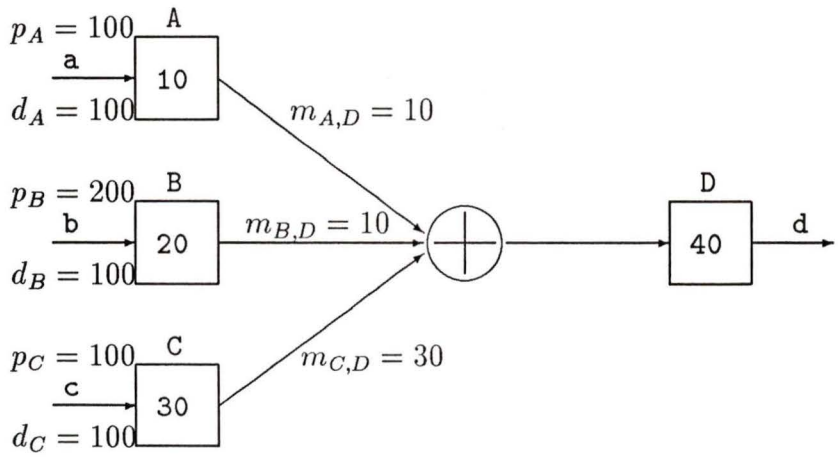



Figure 3.2: **An example of a non-blocking configuration**

the succeeding task. The activated task does not have to wait for completion signals from other predecessor tasks.

#### 3.2.2.1 Non-blocking configuration

Figure 3.2 gives an illustration of a non-blocking configuration of precedence task graph. The figure shows a control system with three input ports and a single output port. There are four function blocks: A, B, C and D. The function block D has three input edges $m_{A,D}$, $m_{B,D}$ and $m_{C,D}$ which are non-blocking as represented by the $\oplus$ sign and a output edge $d$.

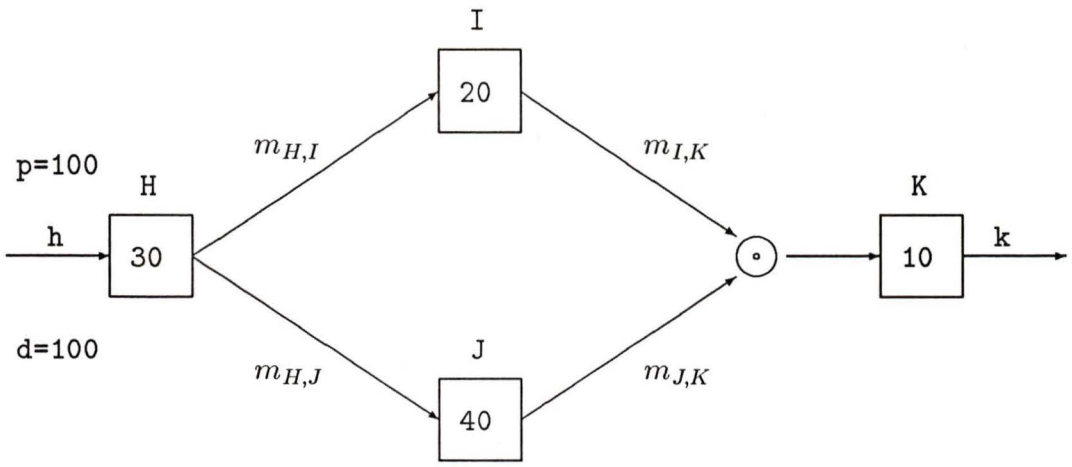



Figure 3.3: **An example of a blocking configuration**

#### 3.2.2.2 Blocking configuration

Figure 3.3 gives an illustration of a blocking configuration of precedence task graph. The figure shows a control system with a single input port and a single output port. There are four function blocks: H, I, J, and K. The function block H has two output edges $m_{H,I}$ and $m_{H,J}$ which are connected to function blocks I and J respectively. The function block K has two input edges $m_{I,K}$ and $m_{J,K}$ which are blocking (i.e., function block K can only proceed after it has received the completion signals from function block I and J), as represented by the $\odot$ sign.

## 3.3 Process synchronization

### 3.3.1 Monitor

To deal with access to critical sections, we need a special process called a monitor which performs some service for ordinary processes on demand. Our monitor is an adaptation from [Mok84a], which is also a simplified version of Hoare's concept [Hoare74] and has the following syntax.

```
Monitor <monitor_name>
Attribute of monitor
{
<A block of codes with no communication primitives>
}
```

The body of a monitor consists of a single scheduling block which is prefixed by a rendezvous primitive with any process as the target. An ordinary process requests service from a monitor by attempting to rendezvous with the monitor. If two or more processes are requesting service, the system scheduler is free to choose a single process to rendezvous with the monitor.

### 3.3.2 Process communication

Our proposed process communication is based on message passing among processes. For brevity, we used *rendezvous*(as defined in Ada's terminology [Ada80]) for inter-process communication.

#### 3.3.2.1 Conditions for rendezvous

In our model, the rendezvous between processes must satisfy the following conditions:

1. The rendezvous between processes take a finite amount of time which in our case is the worst-case communication time delay between the two communicating processes. This can be achieved by using a deterministic channel access approach [Fort86].

2. The rendezvous primitive is used for synchronizing between two processes. More specifically, a rendezvous establishes a precedence constraint which requires that all the computations before the rendezvous primitive in one process must precede all the computations after the corresponding rendezvous primitive in the other process.

3. The rendezvous primitive by itself does not guarantee mutual exclusion (e.g., it should not be used to manipulate sets of variables for which some mutual consistency constraint must be maintained).

4. Rendezvous between a periodic process and a sporadic process is not permitted, since a periodic process must be executed regularly while by definition, there is no guarantee that a sporadic process will request computation at all. If a periodic process must communicate with another process, then the other process must be made ready regularly and is no longer sporadic.

5. If two processes are related via rendezvous primitive, then they must be compatible. Two periodic processes are defined to be compatible if they have the same period or if one period is an exact multiple of the other.

#### 3.3.2.2 Communication primitives

The primary purposes of communication primitives are for passing information between processes and for coordination between scheduling blocks. In our case, the communication primitives are also used as scheduling tools for assignment of tasks with communication constraints. This is vital because in our environment we are assuming a multiple access shared media such as the *Ethernet* [Metcalf76] for communication between processes residing in different nodes.

#### 3.3.2.3 Proposed rendezvous mechanism

We propose a rendezvous mechanism with the following features:

1. A process sends and receives via primitives *send* and *receive*. Having two primitives allow for asymmetric rendezvous which may be useful in a real-time environment.

2. The *receive* primitive can be blocking or non-blocking, but the *send* is only non-blocking. Specifically, the blocking only occurs at the receiving station and not at the sending station.

3. The *send* primitive is associated with two parameters: *deadline* and *transmission time* of the message.

4. The use of input and output process ports for communicating with real-time tasks.

**Receive :** The *receive* primitive has the following syntax:

*receive*("process_id", msg_id, message, **BLK/NBLK**);

where process_id: is the id of the sending process;

msg_id: is the received message_id;

message: is the message received;

**BLK/NBLK**: can either be blocking, **BLK** or non-blocking, **NBLK**.

- *receive* primitive with blocking.

  The receive operation will be blocked if no message has arrived since the last receive. The operation is useful for synchronization of processes.

  Example:

```
Process A()
{
int msg_id;
char *message;
⋮
receive("X", msg_id, message, BLK);
⋮
}
```

- *receive* primitive with non-blocking. The receive operation does not get blocked instead it returns a previously read message. The operation is useful when a task is activated by more than one precedence task or external port. It is also useful where a message needs to be polled periodically but new data is not necessaryly expected to be available at each polling.

  Example:

```
Process B()
{
int msg_id;
char *message;
⋮
receive("Y", msg_id, message, NBLK);
⋮
}
```

- *receive* primitive with many-to-one capability. The receive operation can be either one-to-one or many-to-one. In the many-to-one situation, a process will rendezvous at the receiving end on a first-come-first-served basis. This is useful when a task may be activated by one of several possible precedence tasks or external port. In this case, the *msg_id* may be used as an attribute to identify the sending process.

  Example:

```
Process B()
{
int msg_id;
char *message;
⋮
receive(ANY, msg_id, message, BLK);
⋮
}
```

**Send :** The *send* primitive has the following syntax:

*send*("process_id", msg_id, message, $L_m$, $D_m$);

where process_id:is the id of the destination process;

msg_id: is the message_id;

message: is the message send;

$L_m$: is the length of the message, which is also the worst-case communication time for transmission;

$D_m$: is the deadline of the message.

- *send* primitive with message length and deadline. The send operation provides parameters for specifying the length and deadline of a message.

These parameters are generated after the tasks have been assigned to the respective nodes during the task allocation process.

Example:

```
Process X()
{
char *message;
⋮
send("A", 123, message, 20, 80);
⋮
}
```

These additional parameters: message length, $L_m$, and message deadline $D_m$ are needed because of the following reasons:

1. to prevent priority inversion caused by first-come-first-served protocol, used commonly in multiple access shared media [Locke88],
2. to ensure that the feasible task assignment schedule generated during configuration time will meet timing constraints during run-time and,
3. for the implementation of minimum laxity scheduling of the communication time-slots at run-time. During run-time, the system could implement a modified *virtual-time/CSMA* protocol to allocate the communication time-slots for the multiple access shared media [Molle85].

## 3.4 The real-time scheduling problem

Most deterministic scheduling theories are based on a parameterized model and deal with tasks that are performed only once, i.e., each task is scheduled once and after completion, it is never considered again [Deplan87]. These results are not directly applicable to our problem which differs from them in two essential aspects: our

tasks(processes) may be invoked an infinite number of times, and the request-times of sporadic processes are not known *a priori*. Nevertheless, we can use some of the already developed techniques if we can assume processes are periodic, in which case it is sufficient to examine schedules of length on the order of the least common multiple of the periods.

In general, a real-time scheduling problem involves two schedulers: an off-line or static scheduler and a run-time scheduler. The static scheduler examines an instance of the process model and creates a run-time schedule together with a database for making scheduling decisions at run-time. The run-time scheduler is the code for allocating resources in response to requests generated at run-time, e.g., by timer or external device interrupts.

For single processor, [Mok78] showed the existence of a totally on-line, optimal run-time scheduler for the case where interprocess communication primitives did not impose any scheduling restrictions. In the case where there are communication and mutual exclusion constraints, [Mok84b] shows that it is impossible to find a totally online run-time scheduler.

### 3.4.1 The static schedule

In the context of static schedule, the real-time behavior of a task is described by its worst-case execution time, activation times and deadlines. Since the activation times of a periodic task are deterministic, its real-time behavior can be described precisely. The activation times of a sporadic task however, are not known *a priori*. In [Mok84b], Mok presented a solution called *latency technique*, which may be used to transform a sporadic task into a 'pseudo' periodic task. However it suffers a setback for having too many possibilities in the context of static schedule and a database of finite schedule

computed off-line has to be provided [Mok83].

We propose a transformation of the sporadic task into an 'equivalent' periodic task which behaves exactly like a regular periodic task. However, our method is somewhat pessimistic for tasks with small scheduling slack times[2].

In order to keep the analysis as simple as possible, the effects of blocking on the real-time behavior of tasks in a static schedule is not considered initially. However these effects will be discussed later in Section 3.5.

Condition for scheduling in this model:

> A set of tasks is schedulable if at request-time $t$, the task $(c, p, d)$ is executed completely on a processor for $c$ units of computation time in the interval $[t, t + d]$.

### 3.4.2 Scheduling of periodic tasks

The real-time behavior of a periodic task is determined by its period, the deadline and the computation time. The activation times $X_k$ of a periodic task with period $p$ are given by the following formula:

$$X_k = (k - 1) \times p + t_s$$

where

$k \in$ *integers greater than zero* and
$t_s$ is the time of the task's first activation.

[2]Definition: *scheduling slack time = deadline - computation time*

The time, $t_s$ is the same for all periodic tasks executing on the same physical node but may not be the same for tasks executing on different nodes. The set of all periodic tasks activations is the time interval: $[t_s, t_s + LCM]$, where $LCM$ is the least common multiple of all the periods of all periodic tasks executing on the same physical node. For our analysis, $t_s$ for a specific physical node will be assumed to be zero.

### 3.4.3 Scheduling of sporadic tasks

The activations of a sporadic task can occur at any time with the restriction that successive activations be at least $p$ time units apart, where $p$ is the minimum period of the task activation. For our purposes, the sporadic task is transformed into an 'equivalent' periodic task whose period is small enough to guarantee that an activation of the sporadic task occurring at any time will get enough CPU and communication time slots for it to meet the deadline.

A sporadic task $T$ with a minimum period of activation $p$, a computation time $c$ and a static deadline $d$ may be represented by an 'equivalent' periodic task $T'$ with the following attributes:

**computation time:** $c' = c$,

**deadline:** $d' = \frac{d+c}{2}$,

**period:** $p' = d'$.

We will attempt to show how the attributes of the 'equivalent' periodic task are derived and prove that

"if every activation of this 'equivalent' periodic task can be guaranteed to meet its deadline, then an activation of the actual sporadic task occurring at any time, will be accommodated within its deadline by the execution of one or two 'equivalent' periodic task activations".

Figure 3.4 illustrates two back-to-back task activations (i.e., X and Y) of an 'equivalent' periodic task that are executed as far apart in time as possible. Task X is executed immediately after being activated, whereas task Y is executed just in time to meet its deadline. Given these two extreme activations, the attributes of the 'equivalent' periodic task can be derived as shown in Figure 3.4.

It can be shown that if the actual activation of the sporadic task occurs at any time between the activation time and deadline of X, then its execution time will be accommodated within its deadline by the executions of X and/or Y.

**Proof:** The worst case occurs when the request-time of the sporadic task $T$ occurs immediately after the latest instance of its 'equivalent' periodic task has completed. By definition of the transformation, the next request-time of task $T$ occurs at most $c'$ later. Hence the next instance of sporadic task $T$ is completed at the most $(2 \times p' - c')$ which is $2(\frac{d+c}{2}) - c = d$. □

This technique for scheduling sporadic task is pessimistic in that it schedules more execution time for a sporadic task than it could possibly require at run-time. This is because both the period and deadline of the 'equivalent' periodic task are determined strictly as a function of the scheduling slack time of the sporadic task. As a result, the minimum activation period sporadic task does not enter into the determination of the 'equivalent' periodic task specification.

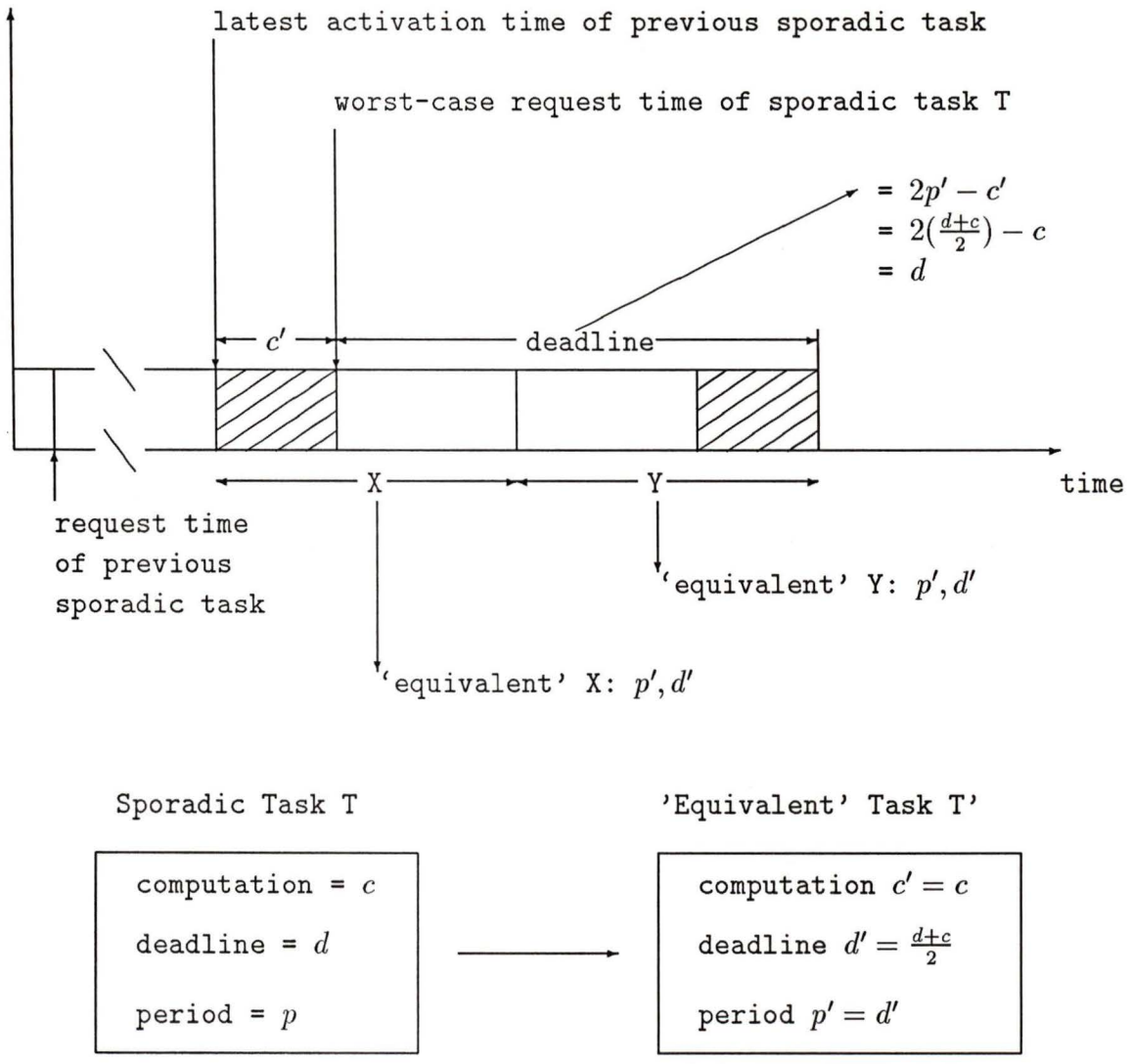



Figure 3.4: **An 'equivalent' task for a sporadic task**

### 3.4.4 Scheduling of monitor

In this model, we adopt a monitor to enforce mutual exclusion, and we allocate processor time only in uninterruptible quantums, say of size $q$ which is chosen to be bigger than the longest monitor. This restriction seems reasonable if critical sections are kept very short. As far as scheduling problem is concerned, the only difference is that a task may be interrupted only after it has been allocated an integral number of time quantums.

## 3.5 Run-time environment

The analysis that was presented in the previous section does not take into consideration the fact that real-time tasks may block during execution. The ability of a real-time task to block during execution has two important effects on the real-time behavior of a processor node. Specifically, whenever a real-time task blocks,

1. The slack times of all tasks succeeding the blocked task in the same task group are shortened.

2. Other real-time tasks with less urgent real-time constraints may execute.

These two effects must be taken into consideration if a static schedule is truly representative of the run-time behavior.

### 3.5.1 Run-time schedule

The first effect is accounted for in a static schedule by adjusting the slack time of any real-time task whose execution may be affected by the blocking of the process

which defines it. The *worst-case* situation is assumed to be one in which a blocking task is just able to meet its deadline. As a result, whenever a task is found to be capable of blocking, the slack times of all succeeding tasks are reduced by a value which is equal to the blocking task's static deadline. For this situation to be realistic, tasks should never block for messages from tasks with later deadlines and time delays should always complete within their specified time limits.

#### 3.5.1.1 Consideration for periodic tasks

For a periodic task, this reduction in slack time is implemented by shifting the activation times of all periodic tasks forward in a static schedule. The amount to be shifted is equal to the static deadlines of the predecessor tasks in the same task group that are capable of blocking. If none of the tasks which precede a task in a group periodic tasks can block, then the activation times of the task are unaffected. Since the absolute deadline of a task activation is not affected by this modification, its scheduling slack time is effectively decreased. The above solution is implemented in the task allocation module which we will discuss in the next chapter.

### 3.5.2 Run-time access of communication channel

In a distributed real-time system, messages as well as tasks must have explicit timing constraints. Furthermore, the most common communication network used in distributed hard real-time systems is the *Ethernet*. In this type of network, stations transmit messages via a shared channel. Only one message can be successfully transmitted on the channel at any one time. No message can be received correctly in the event of a collision.

Various run-time protocols for multiple access networks have been proposed and

evaluated, however most of them may not be suitable for hard real-time communications because the message transmission delay cannot be bounded. Molle and Klienrock [Molle85] suggested a new class of *CSMA* protocols called *virtual-time/CSMA* . In this protocol, each node has two clocks. One clock gives the real time, and another gives the virtual time. The virtual clock stops running when the channel is busy, and runs when the channel is idle. When it runs, the virtual clock runs at a higher rate than the real one if it is behind the real clock. A message is sent only when its arrival time is less than or equal to the time on the virtual clock. The major advantages of the *VT/CSMA* protocol are its fairness in transmitting the waiting messages (since it is based on *first-come-first-served* policy), lower collision rate, and better delay-throughput behavior. As we mentioned in Chapter 2, this may not be satisfactory because it may result in priority inversion if the communication protocol does not have the same algorithm used for task allocation.

Zhao[Zhao86] suggested another *virtual-time/CSMA* protocol for hard real-time communication. Instead of a virtual clock which runs on first-come-first-served, he suggested that the virtual clock should run according to the minimum laxity of messages, which he calls it *VT/CSMA-L* protocol. In this manner, a network wide minimum-laxity-first transmission policy can be achieved. In fact, given a set of messages with deadlines and a feasible schedule for transmitting these messages, a modified *VT/CSMA* protocol can be used to ensure that the messages are received within the bounded times i.e., their deadlines. A feasible schedule of messages with deadlines are discussed in the Section 4.4.1.

### 3.5.3 Modified VT/CSMA protocol

In Section 3.3.2, we proposed using a *rendezvous* with message length and deadline. Using the following definitions, we can transform the earliest-deadline requirement of the message into an equivalent minimum-laxity-first transmission policy.

#### 3.5.3.1 Message transformation

Each message, $m$ can be characterized as follows:

- Length $L_m$, which is the total number of time units needed to transmit message $m$;
- Deadline $D_m$, which is the time by which message $m$ must be received;
- Latest time to send the message, $LS_m$, is equal to $D_m - L_m$;
- Laxity at time t, $LAX_m(t)$, is the maximum amount of time the transmission of message $m$ can be delayed at time t. Therefore,

$$LAX_m(t) = D_m - L_m = LS_m - t.$$

From these definitions, it is clear that transmitting messages according to their latest time to send, is equivalent to the minimum-laxity-first transmission policy.

#### 3.5.3.2 Outline of modified VT/CSMA protocol

The following is the outline of the modified protocol:

- Each node has two clocks, one clock maintaining the real time, and another maintaining the virtual time set in a manner discussed below.

- Each message $m$ is associated with a parameter called *virtual latest time to start transmission*, $VLS_m$. When a message arrives, $VLS_m$ of message $m$ is set to be $LS_m$.

  If the channel is idle and

  $$VLS_{New\ Message} \leq Virtual\ Clock.$$

  Then the new message is sent immediately; Otherwise it waits.

- Each node senses the channel. After a successful message transmission or when the channel is idle, each node resets its virtual clock to be equal to the real clock. The virtual clock starts to run at a rate $\eta > 1$, where $\eta$ is a tuneable parameter of this protocol. The virtual clock stops when the channel is busy.

- A node sends message $m$ when $VLS_m$ is equals to the virtual clock. Note that because the virtual clock runs faster than the real clock, the message is sent at real time,

  $$t + \frac{(VLS_m - t)}{\eta}$$

  where $\eta > 1$. This time is earlier than the latest time to send message $m$, $LS_m$.

- When a collision occurs during the transmission of a message, say at real time $t'$, the sender node

  1. re-transmits this message immediately with probability $P$; or
  2. draws a number $R$ randomly from the interval $(t', LS_m)$, and modifies the virtual latest time to start transmission as,

     $$VLS_m = R$$

3. then, puts this message back in the queue of messages waiting to be transmitted.

- The deadline is missed if the $LS_m$ of a waiting message is less than the current time $t$.

The proof for the original *VT/CSMA* protocol is given in [Molle85]. Our proposed protocol differs in that our scheduling of messages is based on minimum laxity instead of first-come-first-served policy as used in [Molle85].

# Chapter 4

# The Task Allocation System

In this chapter we describe the task allocation system for static assignment of tasks using the graph based model described in the previous chapter. Given an application specified in a graph based computational model, our system tries to allocate tasks among distributed processors to achieve the following goals:

- meet timing and precedence constraints based on the 'worst-case' computational behavior of periodic and sporadic tasks.
- minimize interprocessor communications among tasks.
- maximize parallelism and concurrency of tasks.
- resolve conflicts resulting from accessing shared access communication channels and mutual exclusion constraints.

## 4.1 System structure for task allocation

In order to effectively satisfy these goals, we implemented a system with three supporting functions namely the the task preprocessor module, the task allocation module and the resource allocation module. Figure 4.1 shows the general structure of the system with the three supporting functions.

1. The task preprocessor module takes its input from the specification file provided by the user and performs the following functions:
   (a) Analyzes the 'worst-case' computation requirements of individual task.
   (b) Analyzes the static timing constraints of the computational model by computing the pseudo window for each group of tasks.
   (c) Generates the task priority for each individual task.
2. The task allocation module takes its input from task preprocessor module, and for the given number of processor nodes performs the following functions:
   (a) Creates execution windows for each processor node.
   (b) Selects the available task set for assignment using a heuristic algorithm.
   (c) Analyzes the timing and precedence constraints.
   (d) Allocates communication time delay for tasks with external messages.
   (e) Generates new pseudo windows for all tasks.
3. The resource allocation module using information from both the task preprocessor and task allocation processor, performs the following functions:
   (a) Analyzes any conflict caused by the assignment of communication timeslots to external messages.

(b) Creates a feasible schedule of all external messages.

(c) Resolves any violation of mutual exclusion constraints in each individual processor node.

### 4.1.1 Feasible static schedule

Due to the non-deterministic behavior of sporadic tasks, it is impossible to build a static schedule which describes a node's real-time behavior exactly. The best that can be achieved is to build a pessimistic 'worst-case' schedule.

Assuming that the 'worst-case' computation time and the 'worst-case' communication delay specifications are accurate, the static schedule has the following properties:

1. If the static schedule is found to be feasible then it is guaranteed that a feasible run-time schedule will always exist.

2. If the static schedule is found to be infeasible, then there is a chance that deadlines may be missed at run-time.

3. If the application system consists only of tasks which never block, then the task preprocessor will determine whether a feasible schedule is possible without communication delay.

4. If the application system consists only of tasks which block because of precedence constraints, then the task allocation processor will determine exactly whether the timing constraints can be met.

5. If the application system consists only of tasks which require external messages or shared resources, then the resource allocation processor will determine exactly

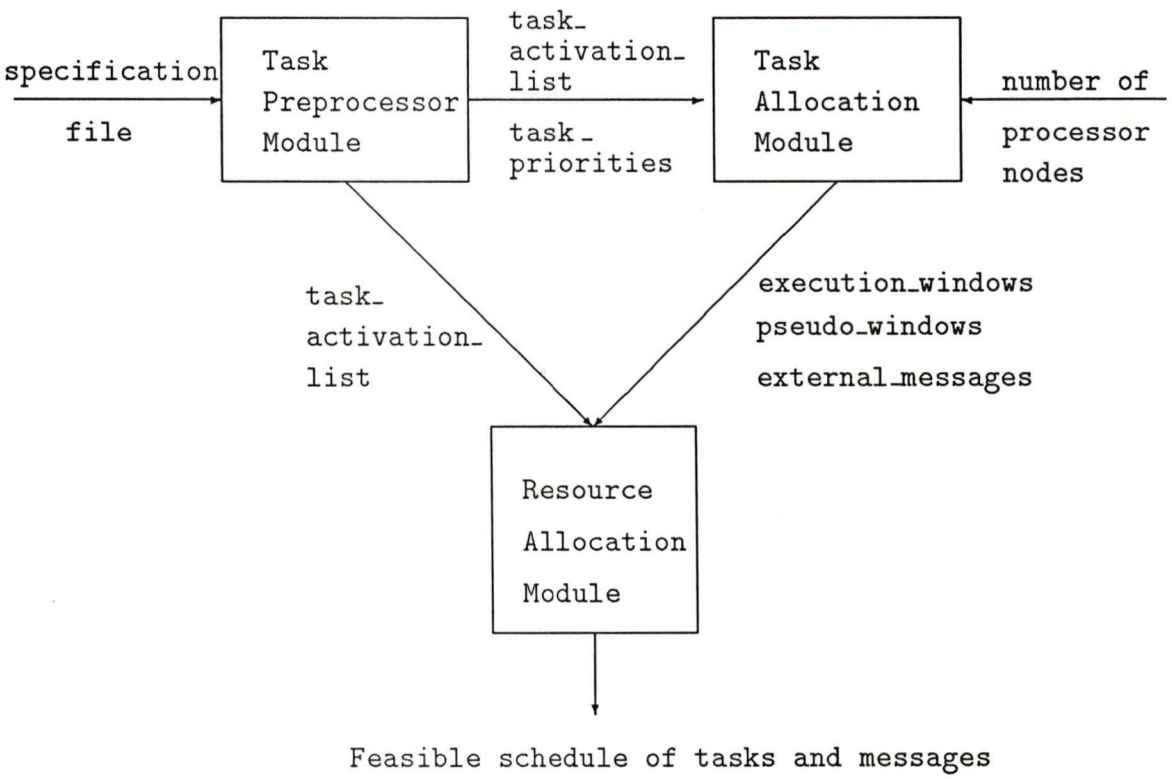



Figure 4.1: **The structure of the task allocation system**

whether the communication requirements and mutual exclusion constraints can be resolved.

## 4.2 Task preprocessor module

In this section we discuss the functions of the task preprocessor module.

### 4.2.1 Worst-case computation time

In this section we analyze the 'worst-case' computation time, *wcct*, of individual tasks. We assume that the behavior of the underlying hardware is deterministic and known. This implies that the timing behavior of all hardware components and the effects of caching and pipelining on task performance are predictable.

The application specific 'worst-case' computation time of a task is the maximum CPU time that the task actually consumes. Trying to get a value for the timing behavior of a task by analysis of its source code[1], one can often derive only the upper bound for the 'worst-case' computation time of a task. This is due to the fact that program code does not contain the full information about the application context of a task.

In order to calculate the 'worst-case' computation time, all parts of a task (i.e., sequences, loops, iterations, etc.) must be computable. This suggests that full information about the control flow and constraints for the control flow in the 'worst-case' have to be known. The main problem is that the control flow of a program at run-time depends on the input data and the current variable settings. The values of variables

[1]This is what can be done by a software tool

used in conditions, and values of pointers to functions, determine the control flow and as a consequence the timing properties of each task. Since it is impossible to simulate the execution of a task for all its possible variable settings, and to determine if the task terminates or how long it takes to execute, some restrictions have to be made in order to get analyzable programs[2].

#### 4.2.1.1 Restrictions for analyzable programs

In order for a program to be analyzable, the following restrictions are required.

- Programs must not contain any recursions. Recursive algorithms have to be replaced by iterative ones or transformed into non recursive schemes.
- Programs must not contains any *goto* statement. Any *goto* construct should be replaced with structure language constructs: *sequences, alternatives, loops and iterations*.
- Calls of subroutines via variables or parameters have to be substituted by explicit subroutine calls.
- Use of *bounded* loops, one that guarantees loop termination within specified amount of time or specified number of iterations.

#### 4.2.1.2 Calculation of 'worst-case' computation time

We used a set of formulas adapted from [Pusch89] to calculate the 'worst-case' computation time of a task. Table 4.1 summarizes the formula used for the calculation of the 'worst-case' computation time.

[2]A detail list of problems related to analyzable programs is given in [Pusch89]

| *construct* | *wcct() function* | *formulae* |
|---|---|---|
| *primitive* | $wcct(primitive)$ | $= \alpha(primitive)$ |
| *sequence* | $wcct(sequence)$ | $= \sum_i wcct(construct_i)$ |
| *alternative* | $wcct(alternative)$ | $= wcct(condition)+$<br>$= MAX(wcct(construct_1), wcct(construct_2))$ |
| $loop_{count}$ | $wcct(loop_{head})$ | $= wcct(init) + wcct(condition)+$<br>$count * (wcct(body) + wcct(condition))+$<br>$wcct(overrun\ statement)$ |
| | $wcct(loop_{tail})$ | $= count * (wcct(body) + wcct(condition))+$<br>$wcct(overrun\ statement)$ |
| $loop_{time}$ | $wcct(loop_{time})$ | $= time + wcct(timeout)$ |

Table 4.1: **Formulas for the calculation of 'worst-case' computation time**

The *wcct()* function is used to calculate an upper bound for the execution time of its argument. The $\alpha$ function takes a *primitive*, (e.g., *malloc*) and returns the amount of time it takes to execute the statement based on the derived number of machine instruction cycles. We shall discuss this function in greater detail in Section 4.2.1.4.

Using the set of formulas listed in Table 4.1, the 'worst-case' computation for a task can be derived as follows:

$wcct(task) = wcct(invocation) + wcct(body)$,
$wcct(body) = \sum_i wcct(construct_i)$,
where $wcct(x)$ is the upper bound of the execution time of $x$.

#### 4.2.1.3 Bounded loops

The construct for bounded loops differ from the usual loop construct in two ways:

- All loop constructs require that a loop bound be specified as suggested in [Pusch89]. A loop bound can either be a limit for the maximum number of iterations or a time limit for the termination of the loop. Loop bounds have to be known at compile time.
- If a loop bound is overrun, the programmer may specify that a specified action be started. The default is the activation of the operating system's exception handler.

The benefit of using bounded loops is twofold. First, they are necessary for the *wcct()* calculation. Second, they serve as a control mechanism for checking iteration limitations at run-time.

#### 4.2.1.4 The $\alpha$ function

The $\alpha$ function is implemented as a software tool for measuring the amount of time required for a primitive to be used in the application context. In order to give a realistic measurement of any given task, we developed the $\alpha$ function on top of a real-time executive using the *NS32016*[3] development board. The $\alpha$ function is implemented by means of two timing primitives in the real-time executive namely, *start_interval_timer* and *stop_interval_timer*. A primitive statement can be measured by means of the following construct:

[3] *NS32016 is a 16-bit wide processor chip manufactured by National Semiconductor*

```
function α (primitive)
structure *primitive;
{
        int x, y;
        x = start_interval_timer();
        execute(primitive);
        y = stop_interval_timer();
        return(y-x);
}
```

### 4.2.2 Task activation paths

In order to identify all possible task activation paths, we have to convert all sporadic tasks into 'equivalent' periodic tasks using the method discussed in Section 3.4.3. The 'equivalent' periodic tasks generated are pessimistic in nature and constitute a 'worst-case' scenario of all task activation paths.

#### 4.2.2.1 Task activation list

For each task activation path, all possible threads of computational tasks are identified and stored in a *task activation list*. Figure 4.2 shows the task activation list for the group of tasks shown in Figure 3.3.

#### 4.2.2.2 Task table

In each task activation list there is a set of *task tables*. Each *task table* identifies a task to be executed within a time frame called *pseudo window*.

We define a pseudo window as a time interval for each task to be scheduled, such that if each task is scheduled within its own pseudo window, then a feasible task

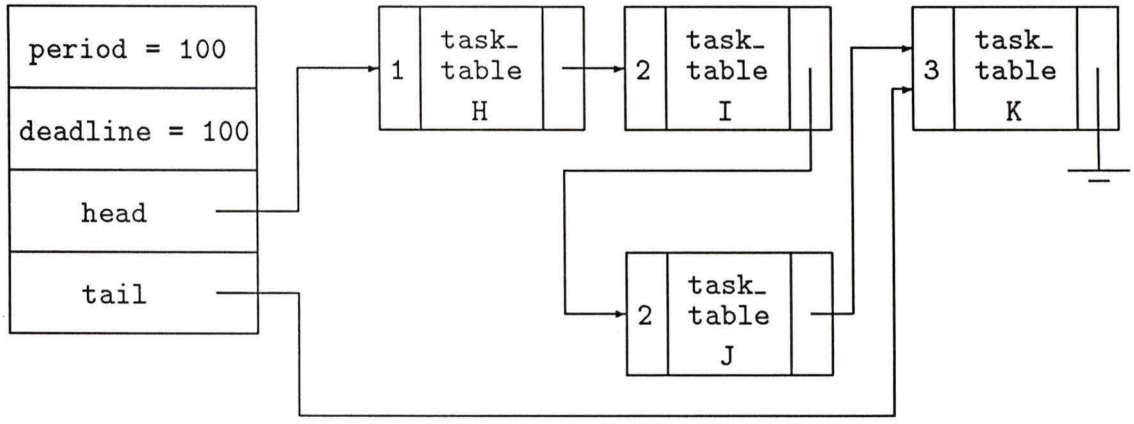



Figure 4.2: **An example of a task activation list**

allocation can be achieved [Cheng86]. A pseudo window consist of a *pseudo ready* and a *pseudo deadline* which represents the feasible execution time span of a task. The pseudo ready and pseudo deadline of a task are dependent on the timing, precedence, communication and other resource constraints. For each task table, a set of message tables are used to represent the communication requirements.

Each message is characterized by the *message id*, *message deadline* ($D_m$), and the *message length* ($L_m$). The message length is the 'worst-case' transmission time between communicating tasks and is bounded. This can potentially be achieved by using a deterministic channel access approach [Fort86]. The message deadline is updated after the external messages have been identified during the task allocation phase. Figure 4.3 shows the task table together with its message tables for task H of Figure 3.3.

### 4.2.3 Sequencing of tasks

The sequencing of tasks within the task activation list is done next. The purpose of sequencing the tasks is to facilitate the computation of the minimum and maximum time delays involved in any task activation path. Furthermore it enables us to differentiate task activations with significant amount of inter-task communications from those task activation with a significant amount of parallelism among tasks.

Tasks in a given task activation list may be ordered in a forward or backward topological order depending on the requirements of the algorithms. Tasks with the same order of sequence(i.e., tasks that can be executed in parallel) are assigned the same *sequence number*. For example in Figure 4.2, since tasks I and J may be executed in parallel therefore they have the same sequence number(i.e., 2).

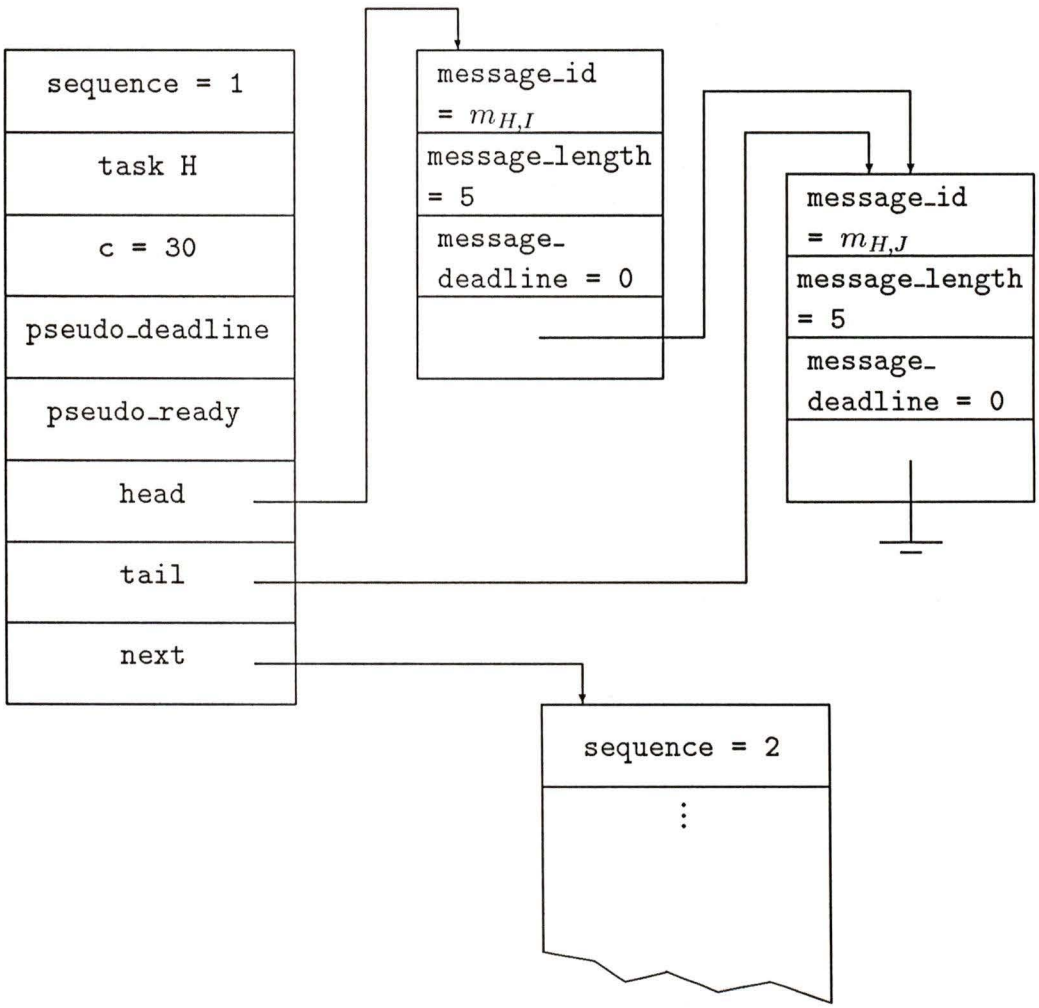



Figure 4.3: **An example of a task table**

### 4.2.4 Computation of pseudo window

The computation of pseudo windows for each task involves computing the the pseudo ready and the pseudo deadline in the task table. The effect of the pseudo window is to assign a pseudo ready time $PR$, and a pseudo deadline $PD$, to a task table. The pseudo ready, $PR$, is the earliest time at which it can be activated, and the pseudo deadline, $PD$, is the latest time by which it must be completed if the precedence constraints or partial ordering of tasks are to be maintained.

#### 4.2.4.1 Algorithm

The algorithm for computing the pseudo ready time and the pseudo deadline for each task is given below.

1. Compute pseudo ready time.

   (a) Sort the tasks generated in the activation task list in forward topological order.

   (b) Initialize the pseudo ready time $PR_{i,k}$ of the $k$ instance of task $T_{i,k}$, in the task activation list to $(k-1) \times p_i$, where $p_i$ is the period of the activated task.

   $$PR_{i,k} = (k-1) \times p_i.$$

   (c) Compute the pseudo ready times of each task in forward topological order by the following formula:

   $$PR_s = MAX\{PR_s, PR_{s'} + c_{s'} + L_{s',s} : s' \rightarrow s\},$$

   where $s'$ is to be scheduled before $s$,

   $PR_s$ is the pseudo ready time of $s$,

$c_{s'}$ is the computation time of $s'$ and

$L_{s',s}$ is the 'worst-case' communication cost between $s'$ and $s$.

2. Compute pseudo deadline.

   (a) Sort the tasks generated in activation task list in reverse topological order.

   (b) Initialize the pseudo deadline ($PD_s$) of the $k$ instance of task $T_{i,k}$, in the task activation list to $(k-1) \times p_i + d_i$, where $p_i$ and $d_i$ are respectively the period and deadline of the activated task.

   $$PR_{i,k} = (k-1) \times p_i + d_i.$$

   (c) Compute the pseudo deadlines of each task in reverse topological order by the following formula:

   $$PD_s = MIN\{PD_s, PD_{s'} - c_{s'} - L_{s,s'} : s \rightarrow s'\},$$

   where $s$ is to be scheduled before $s'$,

   $D_s$ is the pseudo deadline of $s$,

   $D_{s'}$ is the pseudo deadline of $s'$ and

   $L_{s,s'}$ is the interprocess communication cost between s and s'.

Figure 4.4 and Figure 4.5 illustrate the derivation of the pseudo ready time and the pseudo deadline respectively.

### 4.2.5 Pseudo windows with no communication cost

Initially, we construct a set of pseudo windows with no communication cost (i.e., $L_{s,s'} = 0$) and check that the pseudo windows for all the tasks in the task activation list satisfy the condition: $PR_i + c_i \leq PD_i$. If any of the pseudo window fails to satisfy the condition, then we declare failure of static scheduler(i.e., no feasible schedule exist regardless of the number of distributed nodes).

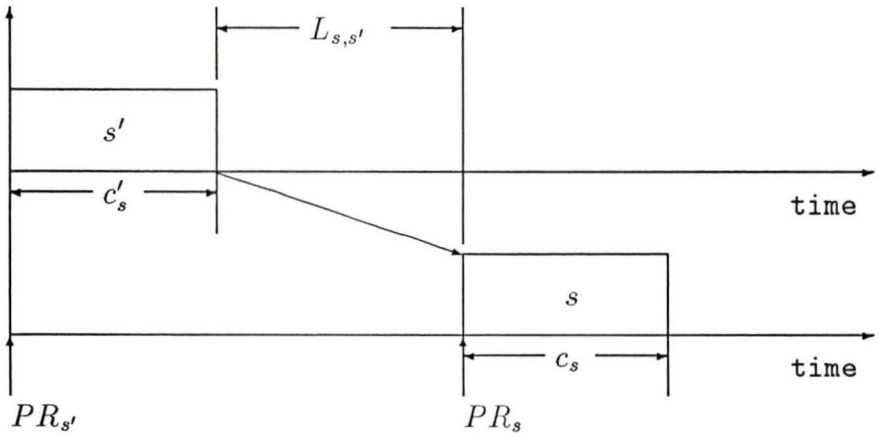



Figure 4.4: **The derivation of the pseudo ready time**

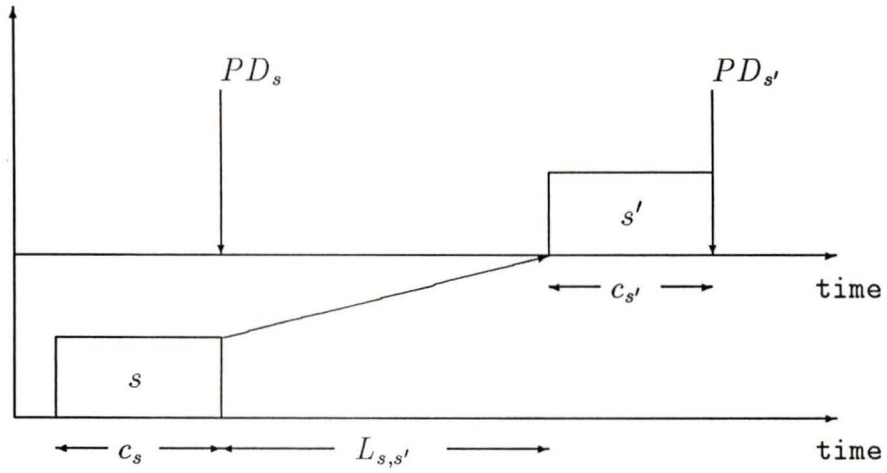



Figure 4.5: **The derivation of the pseudo deadline**

### 4.2.6 Computation of task priority

The *task priority*, $TP_k$, for task $k$ is defined to be the length of the longest path among those connecting the task and the ending task in the group, where the length of a path is the sum of the computation time and the communication time of the tasks in the path.

To compute maximum path length for a sequence of tasks, we use a simple search algorithm to visit every task in the precedence graph according to reverse topological order. The algorithm is given below.

1. Sort the sequence of tasks generated in the task activation list in reverse topological order.

2. For each sequence of tasks, compute the task priority, $TP_k$, in reverse topological order by the formula: $TP_{s,k} = \sum_{i=s}^{n} (c_i + L_i)$ where $c_i$ and $L_i$ are respectively the computation and interprocess communication time for the task $T_i$, $s$ is the sequence number of the task and $n$ is the sequence number of the ending task.

3. Sort the task priority for all the tasks in descending order.

The task priority is used as a secondary factor for determining the priority of allocation of a task during the task allocation phase. The detail for task allocation is discussed in the next section. Figure 4.6 shows the derivation of the task priorities for the four precedence tasks given in Figure 3.3.

## 4.3 Task allocation module

In this section we shall discuss the algorithm for the task allocation module. The purpose of the task allocation is to use heuristic algorithms to allocate a group of

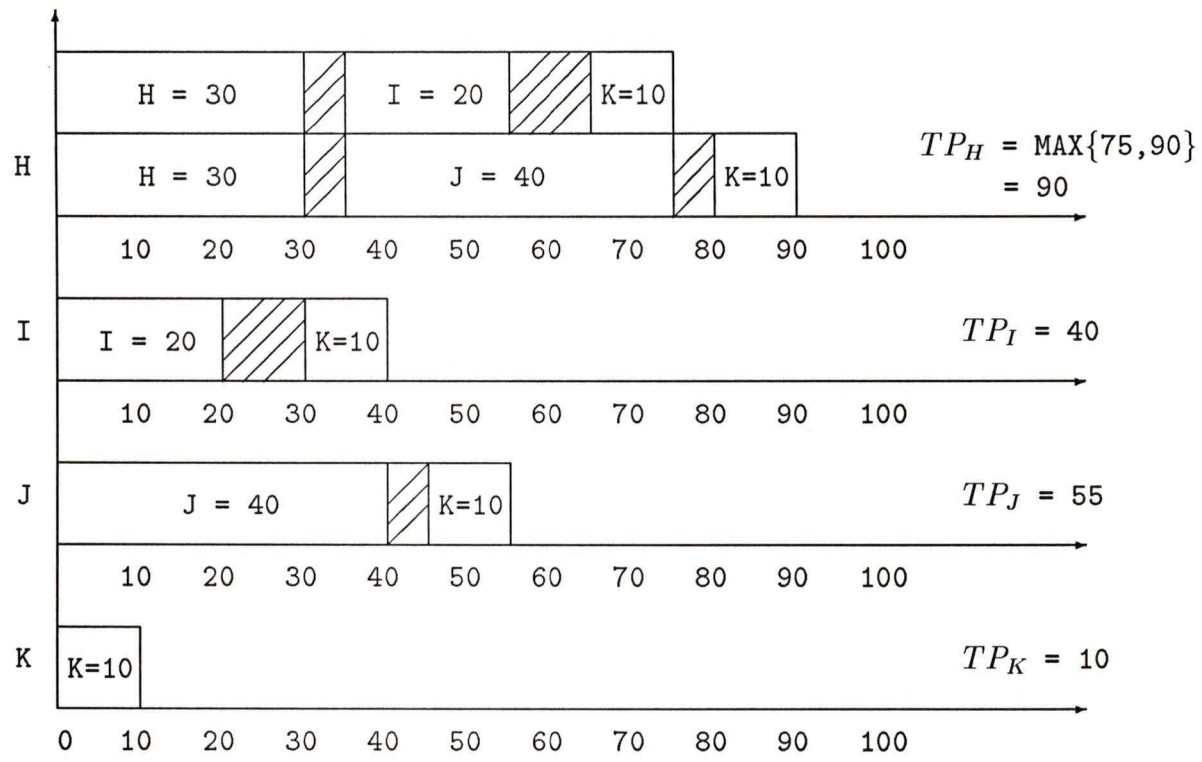



Figure 4.6: **The computation of the task priority**

tasks to a given number of processor nodes. The objectives are

- meet timing and precedence constraints,
- minimize interprocess communications and
- maximize parallel execution of tasks.

### 4.3.1 The basic strategy

The basic functions of the task allocation heuristic is given below.

1. Maintains the earliest time that a new pseudo window can begin in each execution window, called the *current execution window pointer*, $EW$.
2. For each task activation list, identifies the sequence of tasks and maintains an available task set, $ATS$, for task assignment.
3. Computes a temporary pseudo ready time, $TPR$, for each task in the $ATS$, taking into account
   - the available time of a chosen execution window,
   - the communication from predecessor tasks, and
   - the possible clustering of task.
4. Selects the task with the smallest $TPR$ and computes pseudo deadline for the selected task.
5. Appends the pseudo window of the selected task to the execution window chosen for the task and updates the current pointer of the execution window.

6. Updates the available task set, *ATS*.

7. Updates the earliest enable time and latest enable time for each successor task.

This process is repeated until no more task exist in the *ATS*. Initially, the current pointer of each execution window, *EW*, is set to the start time and the size of the execution window, EW_size, is set to the deadline of the of the group of tasks. As each pseudo window is appended to the sequence of an execution window, the current pointer of the execution window is advanced to the selected task's pseudo deadline. The pseudo deadline of a selected task is computed by $PD_s = PR_s + c_s$, where $PR_s$ is the task's pseudo ready time and and $c_s$ is the task's computation time respectively. The pseudo code for the task allocation heuristic algorithm is given in Appendix B.

### 4.3.2 Computing of temporary pseudo window

In order to compute the temporary pseudo windows, first we need to generate an available task set, *ATS*. An available task set is defined as a set of tasks succeeding those predecessor tasks which have been temporarily assigned. Initially, the available task set consists of the beginning task or the first task in the task activation list. When a task is selected for assignment to an execution window, other succeeding tasks of the selected task are considered for inclusion into the available task set. If all the the predecessors of a task have been assigned then the task will be included in the available task set.

The computation of the temporary pseudo ready time of each task in the available task set depends on

1. the execution window the task is appended to,

2. the current pointer of the execution window,

3. whether the task should be clustered with predecessor tasks, and

4. the enable time of the predecessor tasks.

#### 4.3.2.1 The enable time

The enable time is defined as the time at which communication from the predecessor tasks arrives. Furthermore, we define the earliest enable time and latest enable time according to the following equations.

1. the earliest enable time of task $T_j$ is given by,

$$EET_j = MAX\{PD_i\} \tag{4.1}$$

where $PD_i$ is the deadline of a predecessor task, $T_i$.

2. the latest enable time of task of $T_j$ is given by,

   (a) either

$$LET\_1_j = MAX\{PD_i + \sum_{i}^{n} L_{i,j}\} \tag{4.2}$$

when task $T_j$ is not clustered with any of its predecessor task, $T_i$

   (b) or

$$LET\_2_j = MAX\{PD_i + \sum_{i}^{m} L_{i,j}\} \tag{4.3}$$

when task $T_j$ is clustered with some of its
binding predecessor tasks, $BPT_j$

Note that in the equations for $LET\_1_j$, the second factor $\sum_i^n L_{i,j}$, is the summation of communication costs of either all $n$ edges that originated from the same task or all $n$ edges that terminated at the same task. For the case of $LET\_2_j$, the summation is over $m$, where $m = n - k$ and task $T_j$ is clustered with $k$ binding predecessor tasks. However, if any of the predecessor tasks is clustered with a communicating task, the factor is reduced by the corresponding communication cost of the task that has been clustered.

The earliest enable time, $EET_j$, is used whenever task(j) is temporarily assigned to the same processor node as task(i). The latest enable time, $LET\_1_j$, is used whenever task(j) is assigned to a different processor node from all its predecessors. On the other hand, the latest enable time, $LET\_2_j$, is used whenever a task(j) is temporarily clustered with its $BPT_j$ but is on a different processor node from the other predecessors.

A binding predecessor task, $BPT$, is defined as a predecessor task with the latest enable time and is selected according to the task clustering guidelines discussed in the next section. If a task is to be clustered with one or more of its predecessor tasks, it must be clustered with the *binding predecessor task, BPT* in order to reduce its ready time. Furthermore, if a task is to be clustered with its binding predecessor task, it is temporarily appended to the execution window which contains the binding predecessor task.

#### 4.3.2.2 Temporary pseudo ready time

The temporary pseudo ready time of a task, $TPR$, is computed to be the maximum of the current available time of the execution window that the task is temporarily appended to, and the the latest enable time among its predecessors. The temporary pseudo ready time for task(j) is given by the following formula.

If task(j) should cluster with its BPT(i)
Then $TPR_j = MAX\{EW(i), LET\_2_j\}$
Else $TPR_j = MAX\{min\_EW, LET\_1_j\}$

The *min_EW* is defined as the minimum of the current pointers for all the execution windows.

#### 4.3.2.3 Selection of task

The task with the smallest temporary pseudo ready time is selected for assignment. Once the task is selected, the pseudo deadline for the selected task is computed and its temporary pseudo window is appended to the current pointer of the chosen execution window. If more than one task in the available task set have the same temporary pseudo ready time, then the one with the highest task priority $TP$, is selected. The task priority for each task is generated by the task preprocessor module.

At this stage, if the temporary pseudo window of the selected task is within the boundary of the execution window then,

1. update the task table's pseudo window with the temporary pseudo window;

2. update the execution window current pointer, *EW* in which the selected task is appended;

3. update the available task set, *ATS*;

4. update the earliest enable time *EET* and the latest enable times, *LET_1* and *LET_2* for each successor task.

Otherwise the task allocation fails and no feasible schedule can be found.

### 4.3.3 Task clustering guidelines

Our approach is to create clusters which incorporate heuristics to minimize interprocess communication and to increase schedulability of a task. In order to minimize interprocess communication, tasks with the maximum communication costs are normally chosen to be clustered together. The process of selecting the binding predecessor task is governed by the task clustering guidelines as described below:

1. To reduce communication time between tasks, as many tasks as possible should be clustered with their binding predecessor tasks.

2. If the current pointer of the execution window which contains the binding predecessor task is greater than the maximum enable time of the task to be scheduled, then the task should not be clustered with its binding predecessor task, because clustering with the predecessor task cannot reduce its pseudo ready time.

## 4.4 Resource allocation module

The resource allocation module consists of two sub-module namely, communication time-slot sub-module and mutual exclusion sub-module.

### 4.4.1 Communication time-slot sub-module

The communication time-slot sub-module is responsible for the assignment of communication time-slots to all external messages generated by the task allocation module. External messages are assigned to communicating tasks which reside on different processor nodes. Given the assigned time-slots for external messages, the system verifies the feasibility of each assigned time-slot and updates the actual deadline of the message in the task table.

#### 4.4.1.1 The communication time-slot algorithm

The algorithm involves the following steps.

1. Initialization of the communication window and the message queue.
   Initially, the current pointer of the communication window, $CW(t)$ is set to zero and the size of the communication window, $CW\_size$ is set to the deadline of the group of task. All external messages generated in the task allocation module are placed in a message queue and sorted in ascending order of the sending tasks deadlines.

2. Selection of a message for assignment.
   The selection of a message is based on the earliest deadline of the sending tasks, $d_m$, for all the messages in the message queue. If there are more than one message, then the one with the minimum message laxity $LAX_m$ is chosen.

3. Assignment of communication time-slots.

   A communication time-slot is delineated by its begin time, $t_1$ and end time, $t_2$. The assignment of a communication time-slot is computed as follows.

   - time-slot$[t_1, t_2]$:

     (1) If $(CW(t) \leq d_m)$

     Then $\{t_1 = d_m,$

     $t_2 = d_m + L_m\}$

     Else $\{t_1 = CW(t),$

     $t_2 = CW(t) + L_m\}$

     (2) $CW(t) = t_2$.

     where $d_m$ is the deadline of the sending task,

     $L_m$ is the transmission time of the message

     $CW(t)$ is the current pointer of the communication window.

4. Verification of the assignment of the communication time-slot.

   In order for the assignment of the time-slot to be feasible, the derived deadline of the selected message, *message_deadline_new*, is compared with the existing message deadline, the *message_deadline_old*, which was generated earlier during the task allocation phase.

   If the *message_deadline_new* is greater than the *message_deadline_old* then no feasible schedule of the communication time-slot is possible and an error message is generated. Otherwise, the new deadline of selected message is updated in the the task_table.

An outline of the time-slot assignment algorithm and the pseudo code for the subroutine *time_slot_algo()* are given in Appendix B.

Figure 4.7 illustrates the assignments of the communication time-slots for a three message queue. In this example, two of the sending tasks, $T_1$ and $T_3$, have the same deadlines but different laxities, $LAX_1$ and $LAX_2$. Since the deadlines of the sending tasks are the same, the one with minimum laxity is selected (i.e., message $m_1$ in Figure 4.7) and the $CW(t)$ is set to $d_1 + L_1$. Message $m_2$, is assigned last since it has the latest deadline among the sending tasks.

### 4.4.2 Mutual exclusion sub-module

In cases where there are more than one tasks requesting access to a shared resource we have to ensure that no two tasks will access the same resource simultaneously. In our model, we adopt a monitor to enforce mutual exclusion by allocating processor time only in uninterruptible quantums, say of size $q$ which is chosen to be bigger than the longest monitor.

#### 4.4.2.1 Algorithm for mutual exclusion

The following algorithm is used to schedule tasks with overlapping pseudo windows that access a shared resource(or mutual exclusion constraints). The aim of the algorithm is to compute all the possible time slots for tasks that access the shared resource within the common time interval.

For each pseudo window with pseudo ready time, $PR_s$, we consider the interval $RS : q > PR_s - RS_s \geq 0$ to be a *reserved slot* in order not to delay scheduling the task $s$ beyond $RS_s + q$, where $RS_s$ is the start of the reserved slot. The set of reserved slots in the interval $[0, LCM]$ is computed recursively by the following steps.

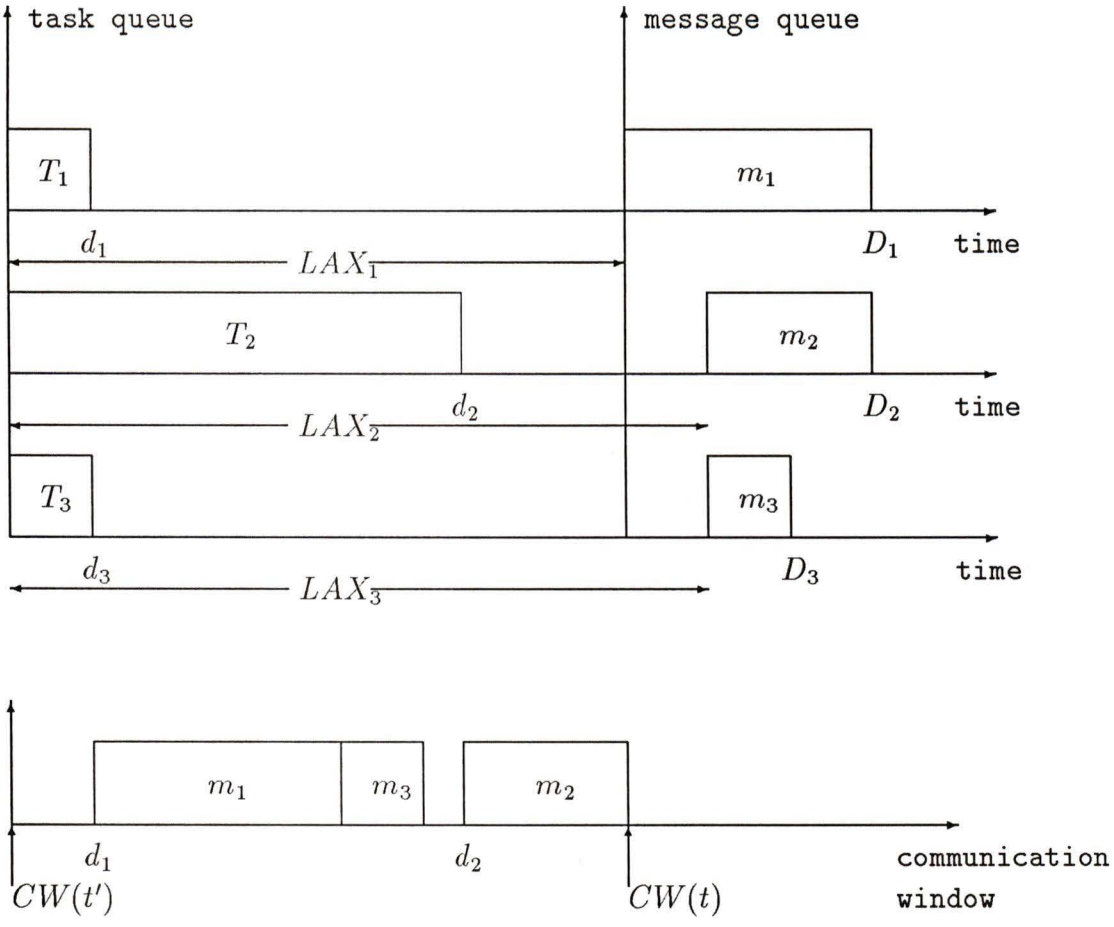



Figure 4.7: **Assignment of communication time-slots**

1. Sort the pseudo ready times in reverse chronological order and determine the reserved slot associated with each pseudo ready time as follows (initially, there are no reserved slots).

2. For each pseudo ready time, $PR_s$, pseudo deadline, $PD_s$, and its deadline $d$ for which $LCM \geq d \geq PD_s$, let $n$ be the number of monitor tasks $s'$ which must be scheduled in the interval $[PR_s, d]$, (i.e., count the number of monitor tasks $s'$ for which $PR_{s'} \geq PR_s$ and $PD_{s'} \leq d$).

3. Given a set of reserved slots in the interval $[PR_s, d]$, schedule $n$ tasks in $[PR_s, d]$ so that none of them starts in a reserved slot.

4. Let $LS_{s,d}$ be the latest time at which the first task must be scheduled. The algorithm starts from time $= d$ and allocates, by moving backward towards $PR_s$, time slots of size $q$ for every monitor task that needs to access the critical section. If placing a time slot $q$ right next to the previous one will result in the left boundary being inside a reserved slot, then the left boundary of the time slot is aligned with the left limit of the reserved slot.

5. If $LS_{s,d} < PR_s$, (i.e., there is no way to fit $n$ tasks of fixed time slot of size $q$ without putting the left boundary of at least one of them in a reserved slot), then declare failure (i.e., no feasible schedule).

   Otherwise, declare $(LS_{s,d} - q, PR_s)$ as the reserved slot if $LS_{s,d} < PR_s + q$.

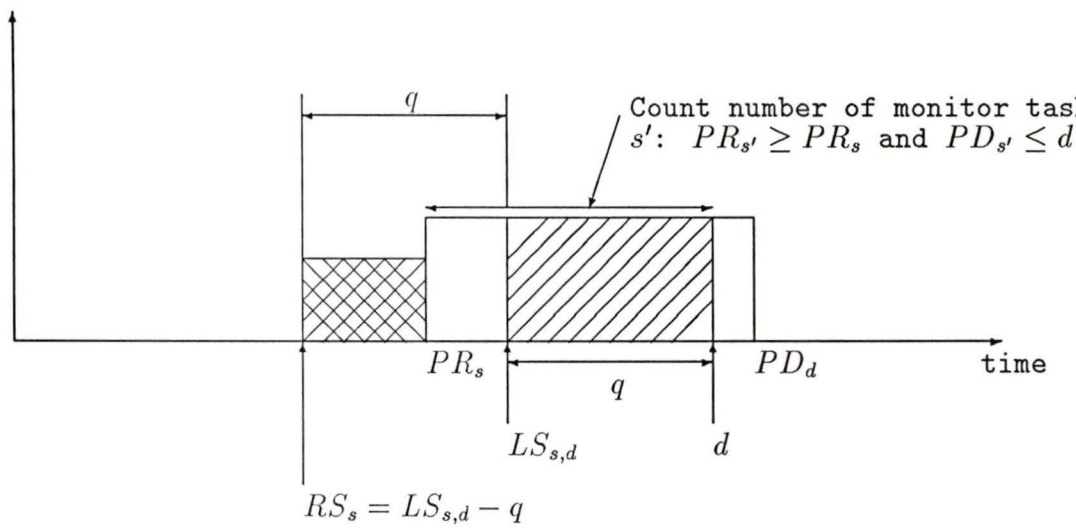
6. Using the above algorithm, we then construct a set of $n$ reserved slots:

   $RS = (pr, pr')$: $pr$ is the revised pseudo ready time
   of some task in $[0, LCM]$, and no task
   should start past $pr$ before $pr'$.

The set $RS$ can be used to locate the reserved slots at run-time as follows:

> At any time t, t lies in the reserved slot,
> iff $pr < MOD(t, LCM) < pr'$ for some $(pr, pr') \in RS$ ,
> where $MOD(x, y)$ is $x$ modulus of $y$.

Figure 4.8 explains how the set of reserved slots are derived for the interval $[PR_s, d]$. The above algorithm ensures that if there are $n$ monitor tasks trying to access the critical region within the specified interval, then there must be at least $n \times q$ quantum amount of time within the interval $[PR_s, d]$ and no process/task should start in the reserved slot region. Otherwise, no feasible schedule exist.




⊠ Reserved slot ⇒ no task should start in this region

Computation of reserved slot

1. n = number of monitor tasks which must be scheduled in the interval $[PR_s, d]$
   = all monitor tasks $s'$ for which $PR_{s'} \geq PR_s$ and $PD_{s'} \leq d$

2. $LS_{s,d}$ = latest time at which the first monitor task should be scheduled

   if $LS_{s,d} < PR_s \Longrightarrow$ no way to fit n tasks within $[PR_s, d]$
   else $RS = (LS_{s,d} - q, PR_s)$ is a reserved slot

Figure 4.8: **The derivation of the reserved slot**

# Chapter 5

# Results and Analysis

In this chapter, we present the results generated by the task allocation system. The process of generating a feasible schedule for a group of precedence tasks involves the following steps.

1. The user provides the specification file for the graph based real-time computation model.

2. The specification file is processed and decomposed into a process-oriented model of precedence tasks.

3. The task preprocessor module generates an initial sets of results for the group of tasks.

4. The task allocation module generates an assignment of tasks with given number of processor nodes.

5. Finally the resource allocation module generates a schedule of the communication time-slot assignments for all external messages. The header file is updated accordingly.

We shall used the example given in Figure 3.1 for the purpose of our discussion and analysis throughout the chapter. Figure 5.1 shows the listing of the run-time execution for the task allocation system. It consists of four parts.

- Part(a) illustrates the invocation of the system and gives a listing of the given specification file.
- Part(b) presents the earliest ready time schedule without consideration for the communication cost involved.
- Part(c) presents the earliest ready time schedule with the worst-case communication delay.
- Part(d) presents the feasible schedule of tasks and messages with three available processor nodes.

The specification file which specifies a graph based computational model for Figure 3.1, is provided by the user. Part(b) and part(c) are generated by the task preprocessor module, while part(d) is generated by the task and resource allocation modules.

## 5.1 Processing the specification file

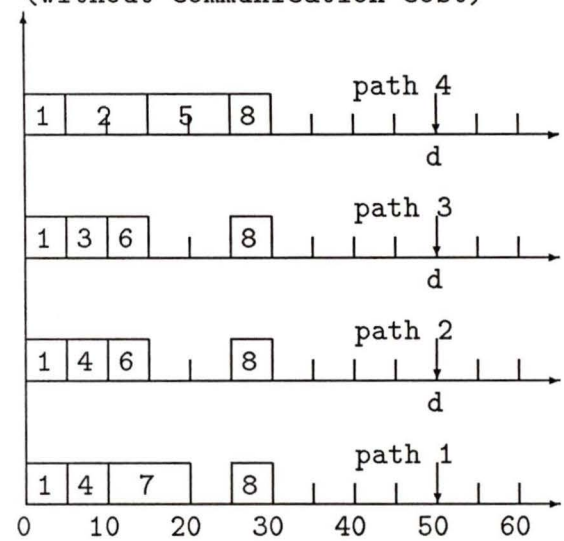
When a specification file for a graph based real-time computation model is created, the file is translated by the specification module into a process-oriented model of precedence tasks.

If the translation is successful, a set of temporary files containing the *task activation list* and *task priority table* for all the tasks are created. If any of the specifications

```
The Pre-runtime system analyzer
Input Specification file > test
%%Node specification
T1 FT1(t1) 5
T2 FT2(m12) 10
T3 FT3(m13) 5
T4 FT4(m14) 5
T5 FT5(m25) 10
T6 FT6(m36,m46) 5
T7 FT7(m47) 10
T8 FT8(m58,m68,m78) 5
%%Edge specification
m12 T1 T2 10 BLK int
m13 T1 T3 5 BLK int
m14 T1 T4 10 BLK int
m25 T2 T5 10 BLK int
m36 T3 T6 10 BLK int
m46 T4 T6 5 BLK int
m47 T4 T7 20 BLK int
m58 T5 T8 10 BLK int
m68 T6 T8 10 BLK int
m78 T7 T8 5 BLK int
%%Input node specification
T1 p 50 100
%%Output node specification
T8 t8 = FT8(m58, m68, m78) int
```

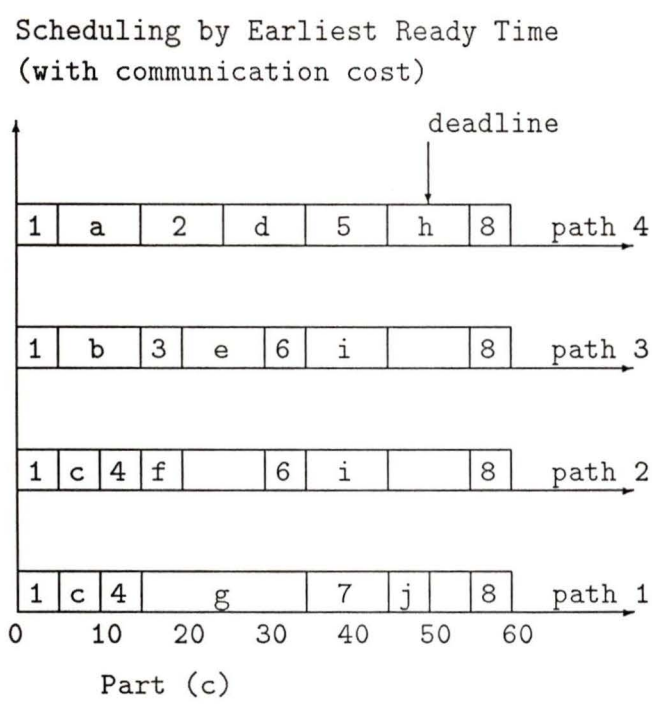
Part (a)

```
Checking the system specification
Please wait for a while ...
Legends :
   !  start/end of a task
   v deadline of task

Symbols :
1:T1    5:T5      a:m12     f:m46
2:T2    6:T6      b:m13     g:m47
3:T3    7:T7      c:m14     h:m58
4:T4    8:T8      d:m25     i:m68
                  e:m36     j:m78

Scheduling by Earliest Ready Time
(without communication cost)
```




Part (b)

Figure 5.1: A run-time execution of the task allocation system



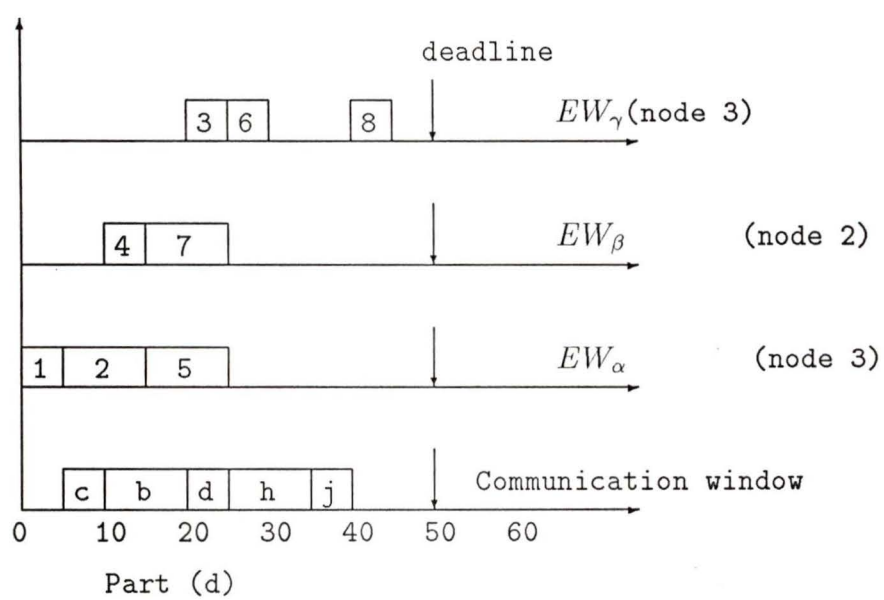




Figure 5.1: **Continued**

are found to be faulty or inconsistent, then the system is aborted with an error message. There are two types of errors that can be detected by the specification module. These are

1. Syntax errors :

   These are errors resulted from wrong syntax of the specification file.

2. Semantic errors:

   A semantic error occurs whenever a specification file is found to be in conflict with the rules which define the behavior and structure of a graph based real-time computational model.

## 5.2 Initial results

Using the information provided in the specification file, the task preprocessor provides an initial analysis of the pseudo window in the task activation list by considering the earliest possible activation time, and computing the task priorities for each individual tasks. All the possible paths in the task activation list are identified and the earliest pseudo ready time are generated for each path.

### 5.2.1 Earliest pseudo ready time

Part(b) and part(c) of Figure 5.1 show the initial analysis generated by the task preprocessor module. Part(b) shows the earliest pseudo ready times in the case where there is no communication delay between tasks. Part(c) shows the earliest pseudo ready times where worst-case communication delays are considered in each activation path.

| task,$T_s$ | maximum path length | task priority,$TP$ |
|---|---|---|
| $T_1$ | {60,50,40,55} | 60 |
| $T_2$ | {45} | 45 |
| $T_3$ | {35} | 35 |
| $T_4$ | {30,45} | 45 |
| $T_5$ | {25} | 25 |
| $T_6$ | {20} | 20 |
| $T_7$ | {20} | 20 |
| $T_8$ | {5} | 5 |

Table 5.1: **The task priority table**

#### 5.2.1.1 Task priority

Using the results generated in part(c) of Figure 5.1, the system generates the task priority for the entire group of precedence tasks. The task priority is the length of the longest path among those connecting the task and the ending task in the group. The task priority for each task is generated by the task preprocessor and used as a secondary factor in determining the selection of a task for assignment (i.e., in cases where there are more than one task with minimum temporary pseudo ready time). Table 5.1 gives the results of the task priority generation for the tasks: $T_1, T_2, T_3, T_4, T_5, T_6, T_7$, and $T_8$ of Figure 3.1.

## 5.3 Allocation of tasks

In this section, we analyze how tasks in the precedence task graph of Figure 3.1 are pre-allocated using our heuristic algorithm. Part(d) of Figure 5.1 shows the final result of the allocation of tasks and messages using three available processor nodes. We shall explain how this result is arrived in the following sections.

### 5.3.1 Execution windows

Since there are three processor nodes (i.e., $\alpha$, $\beta$ and $\gamma$) available for task allocations, the system therefore creates three parallel sequences of execution windows (i.e., $EW_\alpha, EW_\beta, EW_\gamma$). Initially, the size of each execution window, EW_size is set to the group deadline for the group of tasks, and the current pointer of each execution window is set to the start time, $t_s$, which for the purpose of our discussion is equal to zero.

- Window's size: $EW\_size_\alpha = EW\_size_\beta = EW\_size_\gamma = 50$.
- Current pointer: $EW_\alpha = EW_\beta = EW_\gamma = 0$.

The first set of results in part(d) of Figure 5.1 shows the assignment of tasks among the three execution windows, $EW_\alpha$, $EW_\beta$ and $EW_\gamma$. Each task assignment is represented by the task's pseudo window in the respective execution window. Each task's pseudo window is delineated by its pseudo ready time and pseudo deadline.

In the next section, we shall explain how the temporary pseudo windows are generated by the task allocation heuristics.

### 5.3.2 Temporary pseudo windows

Table 5.2 shows the initial results of temporary pseudo windows generated by our heuristic algorithm for the group of tasks given in Figure 3.1. The temporary pseudo window for each task depends on the available task set, the temporary pseudo ready time and the availability of the execution windows.

The heuristics for creating the available task set, computing the temporary ready time, selecting the task in the available task set, the selection of the binding precedence task and the selection of the execution window for assignment are explained in the following sections.

#### 5.3.2.1 Available task set

As defined in chapter 4, the available task set, $ATS$ is a set of tasks whose precedence tasks have been assigned. Initially, the set consists of the beginning task, which is task $T_1$ in our example. One of the task in the set will be selected and assigned to the appropriate execution window.

After the assignment of a selected task, the available task set is updated. For example, after $T_1$ has been assigned, $T_2, T_3$, and $T_4$ become members of this set. The subsequent sets of $ATS$: $\{T3, T_4, T_5\}$, $\{T_3, T_4\}$, ..., are listed in the first column of Table 5.2.

#### 5.3.2.2 Binding predecessor task

The binding predecessor task, $BPT$ is chosen according to the task clustering guidelines described in Section 4.3.3. For example in Table 5.2, $T_1$ is the predecessor task for $T_2$, $T_3$ and $T_4$. According to the task clustering guidelines, both $T_2$ and $T_3$ should

| $ATS$ $T_a$ | possible $BPT$ | latest enable time | smallest $TPR_a$ | selected $T_s$ | window $EW$ | $PD_s$ |
|---|---|---|---|---|---|---|
| $T_1$ | | {0, 0, 0} | 0 | $T_{1,\alpha}$ | $\alpha$ | 5 |
| $T_2$ | $T_{1,\alpha}$ | {5,5} | 5 | $T_{2,\alpha}$ | $\alpha$ | 15 |
| $T_3$ | $T_{1,\alpha}$ | {5,5} | 5 | | | |
| $T_4$ | | {0,30} | 30 | | | |
| $T_3$ | | {0,20} | 20 | | | |
| $T_4$ | | {0,20} | 20 | | | |
| $T_5$ | $T_{2,\alpha}$ | {15,15} | 15 | $T_{5,\alpha}$ | $\alpha$ | 25 |
| $T_3$ | | {0,20} | 20 | | | |
| $T_4$ | | {0,20} | 20 | $T_{4,\beta}$ | $\beta$ | 15 |
| $T_3$ | | {0,20} | 20 | $T_{3,\gamma}$ | $\gamma$ | 25 |
| $T_7$ | $T_{4,\beta}$ | {15,25} | 5 | | | |
| $T_6$ | | {15,40,30} | 20 | | | |
| $T_7$ | $T_{4,\beta}$ | {15,25} | 5 | $T_{7,\beta}$ | $\beta$ | 25 |
| $T_6$ | $T_{3,\gamma}$ | {25,20} | 25 | $T_{6,\gamma}$ | $\gamma$ | 30 |
| $T_6$ | $T_{4,\beta}$ | {25,55} | 55 | | | |
| $T_8$ | $T_{5,\alpha}$ | {25,45,40} | 45 | | | |
| $T_8$ | $T_{6,\gamma}$ | {30,40,40} | 40 | $T_{8,\gamma}$ | $\gamma$ | 45 |
| $T_8$ | $T_{7,\beta}$ | {25,45,50} | 50 | | | |

Table 5.2: **Results of temporary pseudo windows**

be clustered with $T_1$ but $T_4$ should not. Thus $T_1$ becomes the possible $BPT$ of tasks $T_2$ and $T_3$.

Using the clustering guidelines, the possible $BPT$ for each task in the available task set is generated. Therefore, $T_2$ is the possible $BPT$ for $T_5$, and $T_4$ is the possible $BPT$ for $T_7$, and so on.

#### 5.3.2.3 Temporary pseudo ready

In the sample group of tasks, $T_1$ is the predecessor task for $T_2, T_3$ and $T_4$. According the task clustering guidelines, $T_2$ and $T_3$ may be clustered with $T_1$ but not $T_4$. Therefore task $T_1$ becomes the possible $BPT$ of $T_2$ and $T_3$, as shown in Table 5.2 (i.e., rows 2 and 3). After the assignment of task $T_1$ to $EW_\alpha$, the temporary pseudo ready times for tasks in $ATS$ are computed as follows:

$$
\begin{aligned}
TPR_2 = TPR_3 &= MAX\{EW_\alpha, EET_3\} \\
&= MAX\{5, 5\} \\
&= 5. \\
TPR_4 &= MAX\{min\_EW, LET\_1_4\} \\
&= MAX\{0, 5 + (10 + 10 + 5)\} \\
&= MAX\{0, 30\} \\
&= 30.
\end{aligned}
$$

#### 5.3.2.4 The latest enable time

From Section 4.3.2.1, the earliest enable time of a task $T_j$ is given by,

$$EET_j = PD_i,$$

and the latest enable time of task $T_j$ is given by,

- either

$$LET\_1_j = MAX\{PD_i + \sum_i^n L_{i,j}\},$$

- or

$$LET\_2_j = MAX\{PD_i + \sum_i^n L_{i,j}\}.$$

The earliest enable time, $EET$ is used whenever the task is clustered with all its predecessor tasks. The first latest enable time, *LET_1* is used when the task is not clustered with any of its predecessor tasks. The second latest enable time, *LET_2* is use when the tasks is clustered with one or more but not all its predecessor tasks.

The second factor $\sum_i^n L_{i,j}$, used in the computation of *LET_1* and *LET_2*, is the summation of communication cost of either all edges that originated from the same task $T_i$, or all edges that terminated at the same task $T_j$.

For example, there are three communication edges that originate from task $T_1$, therefore $\sum_i^n L_{i,j} = 10 + 10 + 5$. This adjusted factor for communication costs is important because we assumed a shared access communication channel. However, if one of the predecessor is clustered with the communicating task, the factor is reduced by the corresponding communication cost that has been clustered.

For example, since task $T_2$ is clustered with $T_1$, the factor for communication costs is reduced by 10 after the assignment of $T_2$. Hence the temporary pseudo ready time for tasks $T_3$ and $T_4$ are given by,

$$\begin{aligned} TPR_3 = TPR_4 &= MAX\{min\_EW, LET\_2_4 - L_{1,2}\} \\ &= MAX\{0, (5 + 10 + 10 + 5) - 10\} \end{aligned}$$

$$= MAX\{0, 20\}$$
$$= 20.$$

The above heuristic is incorporated for two reasons:

1. to minimize interprocess communication delay by clustering tasks with high communication costs into the same execution window.

2. to increase schedulibity of messages by allocating extra communication bandwidth in cases where are there are higher possibility of using the communication media at the same time.

For example, the heuristic clusters the tasks $T_1, T_2$ and $T_5$ together into $EW_\alpha$ since they have the highest intertask communication delay. In the case of assigning task $T_3$, the pseudo ready time is deliberately set to a later time because there are three messages originating from its predecessor task $T_1$.

### 5.3.3 Selection of task

In selecting the task for assignment, the task with the smallest temporary pseudo ready time $TPR_s$, among the tasks in the available task set is selected. If there are more than one task with the same temporary pseudo ready, then the one with the highest task priority is selected.

The above heuristic is incorporated for two reasons. First, it tries to achieve maximum parallelism by assigning tasks with the smallest temporary pseudo ready time. As shown in the precedence graph of Figure 3.1, tasks $T_2, T_3$, and $T_4$ can possibly run in parallel. This advantage is taken into consideration by the heuristic

algorithm and is reflected in the assignment of the three tasks into three separate execution windows. A similar situation occurred for tasks $T_5, T_6$, and $T_7$ and again the heuristic algorithm assigns them to three separate execution windows as shown in Figure 5.1.

Second, it tries to assign the task with the longest critical path, which is reflected by the task priority. A task with the longest critical path, generally has the highest possibility of missing deadline if scheduled at a later stage. For example, after the assignment of $T_1$, tasks $T_2$ and $T_3$ have the same temporary pseudo ready, $TPR_2 = TPR_3 = 5$. Given that the task priority for $T_2$ is greater than $T_3$(i.e., rows 2 and 3 in Table 5.1), task $T_2$ is selected among the three tasks.

#### 5.3.3.1 Pseudo deadline

Once a task is selected, it is assigned to the execution window and the pseudo deadline is computed as follows.

$PD_j = MAX\{PD_i + L_{i,j}\} + c_j$

where $PD_i$ is the pseudo deadline of predecessor task $T_i$,

$L_{i,j}$ is the communication time between $T_i$ and $T_j$,

$c_j$ is the computation time of $T_j$.

This computed pseudo deadline is used for the computation of the enable time for its succeeding tasks. The time interval from the pseudo ready time to the pseudo deadline becomes the pseudo window of the selected task and is appended on the chosen execution window. Table 5.3 gives the computed pseudo windows of all the eight tasks on the three execution windows, $EW_\alpha$, $EW_\beta$, and $EW_\gamma$.

| task,$T_i$ | $EW_\alpha$ | $EW_\beta$ | $EW_\gamma$ |
|---|---|---|---|
| $T_1$ | 5(0,5) | 0 | 0 |
| $T_2$ | 15(5,15) | 0 | 0 |
| $T_5$ | 25(15,25) | 0 | 0 |
| $T_4$ | 25 | 15(10,15) | 0 |
| $T_3$ | 25 | 15 | 25(20,25) |
| $T_7$ | 25 | 25(15,25) | 25 |
| $T_6$ | 25 | 25 | 30(25,30) |
| $T_8$ | 25 | 25 | 45(40,45) |

Table 5.3: **Assignment of execution windows**

### 5.3.4 Allocation of communication time-slots

After tasks have been allocated to the different processor nodes, the newly derived pseudo windows and message deadlines are updated in the respective task tables. Using this information, the communication time-slots are allocated to all external messages according the heuristic algorithm described in Section 4.4.1.

Table 5.4 shows the message length, the message deadline, and the message laxity for all the messages used in the sample task group. Those communicating tasks which reside on the same processor node will have message lengths and deadlines equal to zero and will not be considered for the allocation of communication time-slots.

| task $T_i$ | message $m_{i,j}$ | length $L_{i,j}$ | deadline $D_m$ | laxity $LAX_m$ |
|---|---|---|---|---|
| $T_1$ | $m_{1,2}$ | 0 | 0 | |
| $T_1$ | $m_{1,3}$ | 10 | 20 | 10 |
| $T_1$ | $m_{1,4}$ | 5 | 10 | 5 |
| $T_2$ | $m_{2,5}$ | 0 | 0 | |
| $T_3$ | $m_{3,6}$ | 0 | 0 | |
| $T_4$ | $m_{4,6}$ | 5 | 25 | 20 |
| $T_4$ | $m_{4,7}$ | 0 | 0 | |
| $T_5$ | $m_{5,8}$ | 10 | 40 | 30 |
| $T_6$ | $m_{6,8}$ | 0 | 0 | |
| $T_7$ | $m_{7,8}$ | 5 | 40 | 35 |

Table 5.4: **Timing constraints of external messages**

| task $T_i$ | message $m_{i,j}$ | laxity $LAX_m$ | task $PD_i$ | time-slot $(t_1, t_2)$ |
|---|---|---|---|---|
| $T_1$ | $m_{1,4}$ | 5 | 5 | (5,10) |
| $T_1$ | $m_{1,3}$ | 10 | 5 | (10,20) |
| $T_4$ | $m_{4,6}$ | 20 | 15 | (20,25) |
| $T_5$ | $m_{5,8}$ | 30 | 25 | (25,35) |
| $T_7$ | $m_{7,8}$ | 35 | 25 | (35,40) |

Table 5.5: **Assignment of communication time-slots**

#### 5.3.4.1 Computation of communication time-slot

Table 5.5 shows the results of the assignment of communication time-slots to the external messages. The messages are sorted in ascending order of deadlines of the sending tasks. The message whose sending task has the earliest deadline is being assigned the communication time-slot. If there are more than one message whose sending tasks have the same earliest deadline then the message with the minimum laxity will be given priority.

For example, in Table 5.5 messages $m_{1,3}$ and $m_{1,4}$ have the same sending task, therefore the same earliest deadline of sending task. However, because the laxity of message $m_{1,3}$ is smaller than that of message $m_{1,4}$, $m_{1,3}$ is given a higher priority.

### 5.3.5 Final result

The second gantt chart in part(d) of Figure 5.1 shows the final results of the assignment of the communication time-slots with respect to the task assignment schedule generated by the task allocation module . In this particular case, since all the external messages can be assigned within the given message deadlines, a feasible schedule of tasks and messages exists.

This clearly shows that our heuristic is able to achieve the following objectives:

1. Meet timing and precedence constraints in a feasible schedule.
2. Minimize interprocess communication.
3. Maximize parallelism and concurrency of tasks.

## 5.4 Comparison of task allocation heuristics

We have used the preprocessing heuristic algorithm given in [Cheng86] to generate a task allocation schedule for the group of precedence tasks in our example, and compared it with the schedule generated by our own heuristic algorithms. We shall call his heuristic algorithm as *Cheng's heuristic algorithm*, in order to differentiate it from our heuristic algorithm.

### 5.4.1 Cheng's heuristic algorithm

Cheng's heuristic algorithm is a preprocessing algorithm for the pre-allocation of tasks in a distributed hard real-time system. It differs from our approach in two ways. First, it does not consider the allocation of sporadic tasks during pre-run-time. Instead, they are considered during run-time through the *focus addressing and bidding process*[Cheng86][Zhao85]. Second, his heuristic does not specifically address issues regarding the contention of communication media. In fact, he assumes that there are unlimited availability of communication channels.

For meaningful comparison, we assume that the number of surplus processors for task allocation is three and the average surplus is one. We further assume that there is no maximum clustering size (i.e., $MCS$ is very large), since the average surplus is one there will not be any difficulty in scheduling large block of task clusters.

Using Cheng's heuristic algorithm, we produced initial results of the temporary pseudo ready times for the group of eight precedence tasks, which are shown in Table 5.6.

### 5.4.2 Assignment of tasks

| candidate task, $T_c$ | latest enable time | smallest $TPR_c$ | chosen $T_s$ | selected sequence | $PD_s$ |
|---|---|---|---|---|---|
| $T_1$ | {0, 0} | 0 | $T_1$ | $\alpha$ | 5 |
| $T_2$ | {5,5} | 5 | $T_2$ | $\alpha$ | 15 |
| $T_3$ | {5,5} | 5 | | | |
| $T_4$ | {0,20} | 20 | | | |
| $T_3$ | {0,15} | 15 | | | |
| $T_4$ | {0,10} | 10 | $T_4$ | $\beta$ | 15 |
| $T_5$ | {15,15} | 15 | | | |
| $T_3$ | {0,15} | 15 | $T_3$ | $\gamma$ | 20 |
| $T_5$ | {15,15} | 15 | | | |
| $T_7$ | {15,15} | 15 | | | |
| $T_5$ | {15,15} | 15 | $T_5$ | $\alpha$ | 25 |
| $T_6$ | {20,20} | 20 | | | |
| $T_7$ | {15,15} | 15 | | | |
| $T_6$ | {15,20} | 20 | | | |
| $T_7$ | {15,15} | 15 | $T_7$ | $\beta$ | 25 |
| $T_6$ | {20,20} | 20 | $T_6$ | $\gamma$ | 25 |
| $T_8$ | {25,35} | 35 | $T_8$ | $\gamma$ | 40 |

Table 5.6: **Results generated by Cheng's heuristic algorithm**

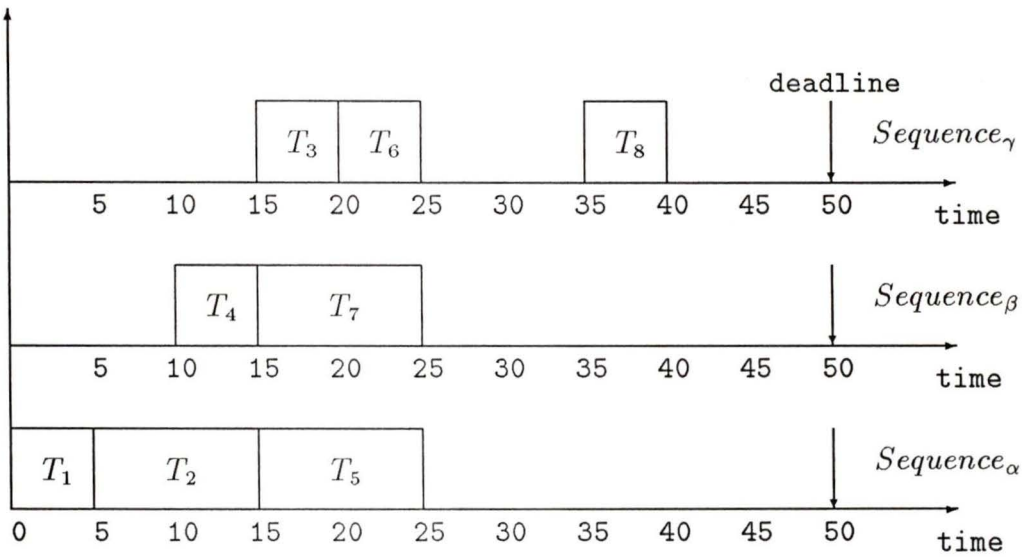



Figure 5.2: The assignment of tasks using Cheng's heuristic algorithm

Figure 5.2 shows the assignment of tasks to three sequence of executions using Cheng's heuristic algorithm. Obviously the heuristic works for meeting deadlines and precedence constraints for all the tasks assigned. However, if we assume that there is only one shared access communication channel for intertask communication, his schedule for the assignment of tasks will fail because of the communication bottlenecks. This will be discussed in the next section.

### 5.4.3 Assignment of communication time-slots

Since Cheng does not have any heuristics for the assignment of external messages, we shall simply use our heuristic algorithm to assign communication time-slots to all external messages created from his task assignment heuristics.

Table 5.7 shows the timing constraints requirement for all external messages in

| task $T_i$ | message $m_{i,j}$ | length $L_{i,j}$ | deadline $D_m$ | laxity $LAX_m$ |
|---|---|---|---|---|
| $T_1$ | $m_{1,2}$ | 0 | 0 | |
| $T_1$ | $m_{1,3}$ | 10 | 15 | 5 |
| $T_1$ | $m_{1,4}$ | 5 | 10 | 5 |
| $T_2$ | $m_{2,5}$ | 0 | 0 | |
| $T_3$ | $m_{3,6}$ | 0 | 0 | |
| $T_4$ | $m_{4,6}$ | 5 | 25 | 20 |
| $T_4$ | $m_{4,7}$ | 0 | 0 | |
| $T_5$ | $m_{5,8}$ | 10 | 35 | 25 |
| $T_6$ | $m_{6,8}$ | 0 | 0 | |
| $T_7$ | $m_{7,8}$ | 5 | 35 | 30 |

Table 5.7: **Timing constraints of external messages**

the task assignment schedule generated by Cheng's heuristics.

Table 5.8 lists the external message queue in ascending order of sending task's deadline and the message laxity. Column 5 shows the possible time-slot assignments. The assignment of time-slot failed for messages $m_{1,4}$ and $m_{7,8}$ because there are more than one messages trying to utilize the communication media at approximately the same time. Since there is no feasible assignment of communication time-slots, the original task assignment will fail during run-time.

| task $T_i$ | message $m_{i,j}$ | laxity $LAX_m$ | task $PD_i$ | time-slot $(t_1, t_2)$ |
|---|---|---|---|---|
| $T_1$ | $m_{1,3}$ | 5 | 5 | (5,10) |
| $T_1$ | $m_{1,4}$ | 5 | 5 | Failed |
| $T_4$ | $m_{4,6}$ | 20 | 15 | (15,20) |
| $T_5$ | $m_{5,8}$ | 30 | 25 | (25,35) |
| $T_7$ | $m_{7,8}$ | 35 | 25 | Failed |

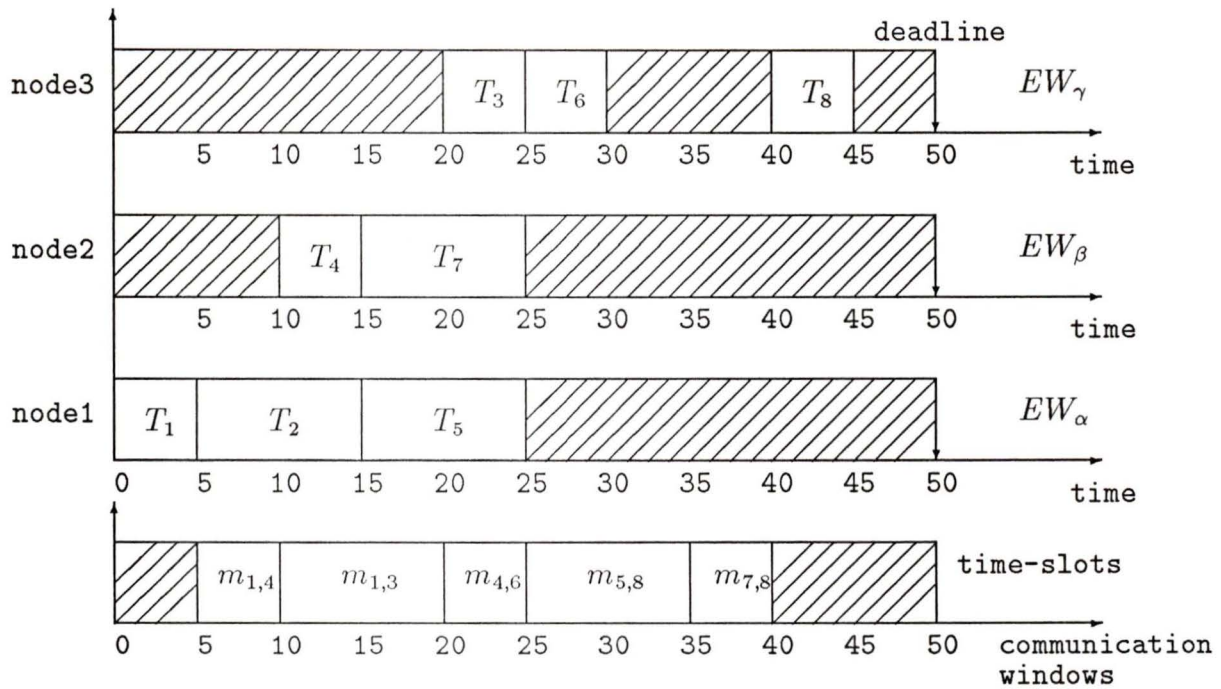
Table 5.8: **Failure in the asssignment of time-slot**

### 5.4.4 Final analysis

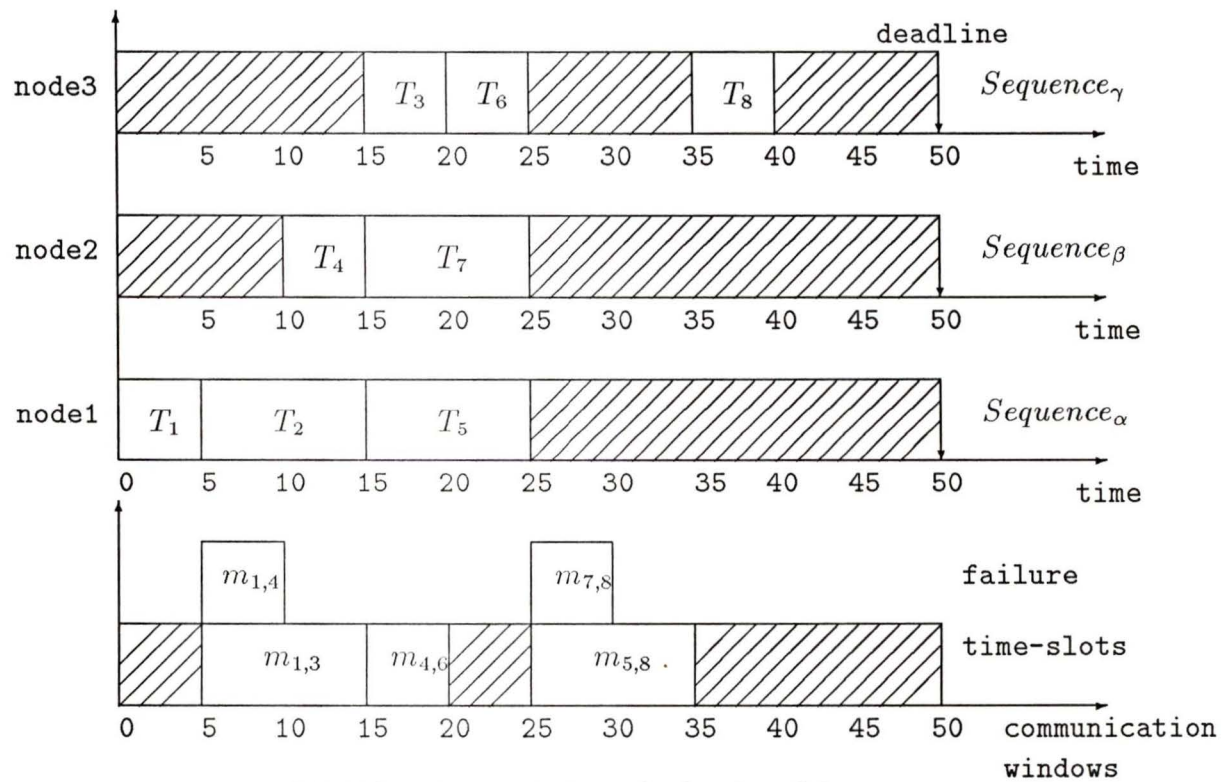
Figure 5.3 shows the complete assignment of tasks and messages derived from the two heuristics. Figure 5.3(a) shows a successful schedule for assignment of tasks as well as communication time-slots, as generated by our heuristic algorithm.

Figure 5.3(b) shows how the attempted assignment of communication time-slots failed for messages originating from the same group of precedence tasks which were scheduled according to Cheng's heuristics.

We have tested our heuristic algorithm on other examples, some of which are given in Appendix B.




(a) Using our heuristic algorithm




(b) Using Cheng's heuristic algorithm

Figure 5.3: The assignment of tasks and messages for the two heuristics

# Chapter 6

# Conclusion

## 6.1 Summary

In this thesis we have proposed a methodology for automating the development of distributed hard real-time systems. We have also introduced and implemented heuristic algorithms which are used in such a development process. The following is a summary of the salient results in this thesis.

1. Designed and implemented a heuristic algorithm for task allocation in a distributed hard real-time environment.

2. Provided a heuristic algorithm for communication time-slot allocation in a *CSMA* environment to meet message deadlines.

3. Proposed a new rendezvous mechanism to meet specific process coordination requirements of distributed hard real-time systems.

4. Provided a method for transforming a sporadic task to an 'equivalent' periodic task.

5. Suggested a run-time environment which integrates the scheduling of tasks as well as messages.

## 6.2 Current status

The task allocation system has been implemented in C and runs on SUN[1] workstations. All the heuristic algorithms presented for the task preprocessor module, the task allocation module and the resource allocation module are included in the implementation, which consists of over 4500 lines of C codes.

We have also developed a system which can translate the graph based real-time system model into a process model of precedence tasks. The system takes the specification file of a graph based model and translates it into a group of processes and monitors where the precedence and timing constraints are embedded in the programs[2].

In general, our system has provided means for specifying, verifying and enforcing timing constraints for hard real-time systems at pre-run-time. It is also a useful tool for analyzing the feasibility of scheduling a group of tasks with a given number of processor nodes using a *CSMA* bus for communication.

In practice, simple resource scheduling disciplines such as round robin or static priority list are often used for resource scheduling in hard real-time environment. While these scheduling mechanisms are simple to implement, they offer only limited and rather inflexible control over response times. Even in recent methods for resource scheduling where heuristics are used, many have failed to address the issue of communication bottlenecks.

[1]SUN is the trademark of *Sun Microsystems*

[2]The detail is given in appendix A

Our system is able to achieve much better results by creating timing and precedence constraint specifications for external messages and using heuristics which specifically take into consideration the communication bottlenecks for the allocation of tasks and messages. This is illustrated in the example presented in Chapter 5. Our heuristic algorithm was able to schedule tasks and messages successfully, whereas Cheng's heuristic algorithm [Cheng86] used for the same group of precedence tasks failed to do so.

## 6.3 Future research

Our efforts have been concentrated more on pre-run-time task allocation schemes, so that a feasible static schedule can be presented to the run-time system. More specifically, we have formulated the scheduling problems so that either a feasible schedule is found or the user is told that a feasible implementation is not possible. It is desirable to be able to give the user more feedback information about the tradeoffs that can be made to arrive at a feasible solution. This is one area that needs further investigation.

Finally, we have not addressed the problems of run-time system failure and recovery which must be dealt with in any hard real-time system. Taking a step further from what we have accomplished, a fault-tolerant run-time system may consist of replicas of our pre-run-time set of nodes doing identical computation and communicating through broadcast calls. This suggestion may also be worthy of further investigation.

# Bibliography


[Abrams70] Abramson, N., "The ALOHA System - Another Alternative for Computer Communications," *AFIPS Proceedings of Fall Joint computing*, vol. 37, 1970, pp. 1476-1979.

[Ada80] "Reference Manual for the Ada Programming Language," *Proposed Standard Document, United States department of Defense*, Jul. 1980.

[Alford77] Alford, M.W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, vol. 3, no. 1, Jan. 1977, pp. 60-69.

[Barnes79] Barnes, R., "A Working Definition of the Proposed Extensions for PL/1 Real-Time Applications," *SIGPLAN Notices*, vol. 14, no. 10, Oct. 1979, pp. 77-99.

[Birman84] Birman, K.P., Joseph, T.A., Rauchle, T., Abbadi A.E., "Implementing Fault-Tolerant Distributed Objects," *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems*, Oct. 1984, pp. 124-133.

[Brinch78] Brinch-Hansen, P., "Distributed Processes; a Concurrent Programming Concept," *Communications of the ACM*, vol. 21, no. 11, Nov. 1978, pp. 934-941.

[Cape79] Capetanakis, J.I., "Generalized TDMA: The Multi-Accessing Tree Protocol," *IEEE Transactions on Communication*, vol. COM-27, 1979, pp. 1476-1484.

[Cheng86] Cheng, S., Stankovic, J.A., Ramamritham, K., "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1986, pp.166-174.

[Chlam79] Chlamtac, I., Franta, W.R., Levin, D., "BRAM: The Broadcast recognizing Access Method," *IEEE Transactions on Communications*, vol. COM-27, No. 8, Aug. 1979 pp. 1183-1189.

[Chu80] Chu, W.W., Holloway, L.J., Lan, M.T., and Efe K., "Task Allocation in Distributed Data Processing," *IEEE Computer*, Nov. 1980, pp. 57-69.

[Cohen78] Cohen, R.M., "Formal Specifications of Real-Time Systems," *Proceedings of the 7th Texas Conference on Computing Systems*, Oct. 1978, pp. 1.1-1.8.

[Cooper84] Cooper, E.C., "Circus: A Replicated Procedure Call Facility," *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems*, Oct. 1984, pp. 11-24.

[Cooper85] Cooper, E.C., "Replicated Distributed Programs," *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Dec. 1985, vol. 19, no. 5, 63-78.

[Davies83a] Davies, D.W., etal., "Motivations, Objectives and Characterization of Distributed Systems," *Distributed Processing - Architecture and Implementation*, Springer-Verlag, Berlin and New York, 1983, pp. 1-9.

[Davies83b] Davies, D.W., etal., "Distributed System Architecture Model," *Distributed Processing - Architecture and Implementation*, Springer-Verlag, Berlin and New York, 1983, pp. 10-43.

[Deplan87] Dephanche, Anne-Marie, Elloy, Jean-Pierre, "Task Redistribution with Allocation Constraints in a Fault-Tolerant Real-time Multiprocessor System," *Proceedings of the IFIP conference 10.3 Working Conference on Distributed Processing*, Oct. 1987, pp. 133-150.

[Dijks68] Dijkstra, E.W., "Cooperating Sequential Processes," *In Programming Language, edited by Genuys, F., Academic Press*, 1968, pp. 43-112.

[Faulk88] Faulk, S.R., Parnas, D.L., "On Synchronization in Hard Real-Time Systems," *Communications of the the ACM*, vol. 31, no. 3, Mar. 1988, pp. 274-287.

[Fort86] Fortier, P., "A Real-Time Distributed Operating System (RTDOS)," *IEEE 3rd Workshop on Real-Time Operating Systems*, Feb. 1986, pp. 3.5-3.12.

[Gehani88] Gehani, N.H., Roome, W.D., "Concurrent C++: Concurrent Programming with Class(es)," *Software Practice and Experience*, vol. 18(12), Dec. 88, pp. 1157-1177.

[Hen80] Heninger, K.L., "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, Jan. 1980, pp. 45-56.

[Hoare74] Hoare, C.A.R., "Monitors: an Operating System Structuring Concept," *Communications of the ACM*, vol 17, no. 10, Oct. 1974, pp. 549-557.

[Hoare78] Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, Aug. 1978, pp. 666-677.

[Klein78] Kleinrock, L., Yemini, Y., "An Optimal Adaptive Scheme for Multiple Access Broadcast Communication," *Proceedings of IFC*, 1978, pp. 7.2.1-7.2.5.

[Klein83] Kleinrock, L., Scholl, M.O., "Packet Switching in Radio Channels: New Conflict-Free Multiple Access Environment," *Proceedings of the 8th IEEE Int. Data Communication Symposium*, 1983, pp. 141-154.

[Kopetz85] Kopetz, A.H., Merker, W., "The Architecture of MARS," *Proceedings 15th Fault-Tolerant Computing Symposium*, Jun. 1985, pp. 274-279.

[Lam80] Lam, S.S., "A Carrier Sense Multiple Access Protocol for Local Networks," *Computer Networks*, vol. 4, Feb. 1980, pp. 22-32.

[Lamport78] Lamport, L., "Time, Clock and Ordering or Events," *Proceedings in the 3rd Principles of Distributed Systems, Communications of the ACM*, vol. 21, no. 7, Jul. 1978, pp. 558-565.

[Lampson83] Lampson, B.W., "Atomic Transactions," *In Distributed Systems - Architecture and Implementation*, Springer-Verlag 1983, Berlin and New York, pp. 246-265.

[Lein82] Leinbaugh, D.H., Yamini, M., "Guarantee Response Times in a Distributed Hard Real-Time Environment," *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1982, pp. 157-169.

[Locke88] Locke, D., Sha, L., Rajkumar, R., Lechoczhy, J.P., Burns, G., "Priority Inversion and Its Control: An Experimental Investigation," *Proceedings of the 2nd ACM International Workshop on Real-Time Ada*, Ada Letters, Special Edition, vol. 8, no. 7, 1988, pp. 39-42.

[Lomet80] Lomet, D., "Subsystems of Processes with Deadlock Avoidance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, May 1908, pp. 297-303.

[Ma82] Ma, P.Y.R., Lee, E.Y.S., Tsuchiya, M., "A Task Allocation Model for Distributed Computing Systems," *IEEE Transactions on Computers*, vol. C-31, no. 1, Jan. 1982, pp. 41-47.

[Magee83] Magee, J., Kramer, J., "Dynamic Configuration for Distributed Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1983, pp. 277-289.

[Metcalf76] Metcalfe, R.M., Boggs, D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, vol. 19, Jul. 1976, pp. 395-404.

[Mok78] Mok, A.K., Dertouzos, M.L., "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proceedings of the 7th Texas Conference on Computing Systems*, Houston, Nov. 1978, 5.1-5.12.

[Mok83] Mok, A.K., "Fundamental Design Problems of Distributed Systems for Hard Real-Time Environment," *Technical Report. MIT/LCS/TR-297*, Ph.D. Dissertation, MIT, 1983.

[Mok84a] Mok, A.K., "The Design of Real-Time Programming Systems Based on Process Models," *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1984, pp. 5-17.

[Mok84b] Mok, A.K., "The Decomposition of Real-Time System Requirements into Process Models," *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1984, 125-134.

[Mok88] Mok, A.K., "What is so Difficult about Time?," *IEEE Real-Time Systems Newsletter*, vol. 4, no. 2, Summer 1988, pp. 14-15.

[Molle85] Molle, M.L., Kleinrock, L., "Virtual Time CSMA: Why Two Clocks are Better than One," *IEEE Transactions on Communications*, vol. COM-33, no. 9, Sep. 1985, pp. 919-933.

[Pusch89] Puschner, P., Koza, C., "Calculating the Maximum Execution Time of Real-Time Programs," *Research Report Nr. 1/89, Institut für Technishe Informatik*, Technishe Universität Wien, March 1989, pp. 1-25.

[Ramam84] Ramamritham, K., Stankovic, J.A., "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software*, vol. 1, 3(Jul 84), pp. 65-75.

[Ramam87] Ramamritham, K., Stankovic, J.A., Zhao W., "Meta-level Control in Distributed Real-Time Systems," *Proceedings of the 7th IEEE Conference on Distributed Computing Systems*, 1987, pp. 10-17.

[Sha86] Sha, L., Lechoczky, J.P., Rajkumar, R., "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *IEEE Real-Time Systems Symposium*, 1986, pp. 181-191.

[Sha88] Sha, L., "Real-Time System Engineering Methodology," *IEEE Real-Time Systems Newsletter*, vol. 4, no. 2, Summer 1988, pp. 12-13.

[Stan88] Stankovic, J. A., "Critical Issues in Distributed Real-Time Operating Systems," *IEEE Real-Time Systems Newsletter*, vol. 4, no. 2, Summer 1988, pp. 16-18.

[Tanen88] Tanenbaum, A.S., "Computer Networks," *Prentice-Hall, Inc.*, 2nd Edition, 1988, pp. 116-195.

[Wei80] Wei, A.Y., Hiraishi K., Cheng, R., Campbell, R.H., "Application of Fault-Tolerant Deadline Mechanism to a Satellite On-Board Computer System," *Digest of papers, The 10th Interbational Symposium on Fault-Tolerant Computing*, Kyoto, Japan, Oct. 1980, pp. 107-109.

[Zhao85] Zhao, W., "Distributed Scheduling Using Bidding and Focussed Addressing," *Proceedings of IEEE Real-time Systems Symposium*, Dec. 1985, 103-111.

[Zhao86] Zhao, W., Ramamritham, K., "A Virtual Time CSMA Protocol for Hard Real-Time Communication," *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1986, pp. 120-127.

# Appendix A

# Decomposition Module

This appendix describes the implementation of the decomposition module and discusses the strategy used for transforming a formal specification of a computational model into a process model.

## A.1 Approach

In our model, after a real-time computational model has been specified, it is decomposed into a process-oriented model. In [Mok84b], Mok suggested three ways of decomposing a computational model namely,

1. decomposition by critical timing constraints,
2. decomposition by minimizing interprocess communication and
3. decomposition by maximizing concurrent processes.

Our approach is an integrated approach, whereby the computational model is decomposed by considering both timing constraints and minimizing interprocess communication. The issue of maximizing concurrent processes is considered only at the end, during the task allocation phase.

The system takes its input from a specification file provided by the user and generates a set of processes and monitors which reflects the timing and precedence constraints of the hard real-time system.

### A.1.1 Specification file

In our system, a real-time computational model is specified through a specification file. The specification file is used for specifying the precedent task graph given in a computational model.

Figure A.1 gives the format of the specification file.

## A.2 Decomposition strategy

In this strategy, a process is created to perform the computation associated with each functional element. A process may consist of a sequence of functional elements with given computation times and precedence constraints. Since a functional element may be called by more than one process, a monitor is created to enforce mutual exclusion on the execution of any function element called by two or more processes. The outline of the decomposition strategy is given below.

1. For each input node $V$, in the communication graph $G$, let $e_{in}$ and $e_{out}$ be respectively its input and output edges. we create a process Q as follows.

**Process** Q()
{
**data-type** $e_{in}, e_{out}$
$e_{out}$ = function_V ($e_{in}$);
**send** ("$S$","$Q$", $e_{out}, L_{S,Q}, deadline$);
}

2. For each node $V$, in the communication graph $G$, which has more than one input edges, let $e_{in,1}...e_{in,n}$ and $e_{out}$ be respectively its input and output edges. We create a monitor $M$ as follows.

**data-type Monitor** M ($e_{in,1} \cdots e_{in,n}$);
**data-type** $e_{in,1} \cdots e_{in,n}$;
{
**data-type** $e_{out}$;
$e_{out}$ = function_V ($e_{in,1} \cdots e_{in,n}$);
}

If the input edges are non-blocking and their periods are compatible, we create a process Q_M as follows.

**Process** Q_M ()
{
**data-type** $e_{in.1} \cdots e_{in,n}$;
**data-type** $e_{out}$;

**If** EVERY ($p_{p,1}, LCM$)
**Then** *receive* ("$R_1$",, $e_{in,1}, BLK$);
**Else** *receive* ("$R_1$",, $e_{in,1}, NBLK$);
⋮
**If** EVERY ($p_{p,n}, LCM$)
**Then** *receive* ("$R_n$",, $e_{in,n}, BLK$);
**Else** *receive* ("$R_n$",, $e_{in,n}, NBLK$);

$e_{out}$ = function_V ($e_{in,1} \cdots e_{in,n}$);
}

If the input edges are blocking , we create a process Q_M as follows.

**Process** Q_M ()
{
**data-type** $e_{in,1} \cdots e_{in,n}$;
**data-type** $e_{out}$;
**receive** ("$R_1$", $e_{in,1}$, , $BLK$);
⋮
**receive** ("$R_n$", $e_{in,n}$, , $BLK$);
$e_{out}$ = function_V ($e_{in,1} \cdots e_{in,n}$);
}

The number of rendezvous,**receive** command in the process Q_M depends on the number of input edges the node V has.

3. For each output node V, in the communication graph G, let $e_{in}$ and $e_{out}$ be respectively its input and output edges. we create a process Q as follows.

**Process** Q ();
{
**data-type** $e_{in}, e_{out}$;
**receive** ("$R$", , $e_{out}, BLK$);
output_V ($e_{out}$);
}

The primitives and functions used in the constructions of process and monitor are given in table A.1

### A.2.1 Example

Using the examples shown in section 3.2, we give an illustration of the decomposed process and monitor. Figures A.3 and A.4 show respectively the created processes

```
%% node specification
<node_id> <function(attributes)> <computation>
  . . .
%% edge specification
<edge_id> <sources> <destination> <L(s,d)> <BLK/NBLK> <data-type>
  . . .
%% input node specification
<start_id> <p/s> <period> <deadline>
  . . .
%% output node specification
<node_id> <output = function(attributes)> <data-type>
  . . .
[EOF]
```

Figure A.1: **Format for the specification file**

and monitor of the decomposed process model for the example specified in section 3.2. Note that in the header section of Figure A.3, the deadlines for the messages: m1_deadline, m2_deadline and m3_deadline are initially set to zero. They will be assigned only if the communicating processes reside on different processor nodes. This is determined during the task allocation phase. The parameters m1_length, m2_length and m3_length are the transmission time of the three messages.

The specification files for the computation models of Figures 3.2 and 3.3 are given in Figures A.2(a) and A.2(b) respectively.

```
%%Node specification
A FA(a) 10
B FB(b) 20
C FC(c) 30
D FD(m1, m2, m3) 40
%%Edge specification
m1 A  D  20 NBLK int
m2 B  D  10 NBLK int
m3 C  D  10 NBLK int
%%Input node specification
A p 100 100
B p 200 200
C s 200 200
%% output node specification
D  d = FD(m1, m2, m3)  int
```

(a) Specification file for figure 3.2

```
%%Node specification
H FH(h)   30
I FI(m1) 20
J FJ(m2) 40
K FK(m3, m4) 10
%%Edge specification
m1 H I  BLK int
m2 H J  BLK int
m3 I K  BLK int
m4 J K  BLK int
%%Input node specification
H p 100 100
%% output node specification
K k = FK(m3, m4) int
```

(b) Specification file for figure 3.3

Figure A.2: Examples of specification files

```
/* header section */
#define period_A 100
#deinfe period_B 200
#deinfe period_C 200
#define LCM      100
#define m1_deadline 0
#define m2_deadline 0
#define m3_deadline 0
#define m1_length   10
#define m2_length   10
#define m3-length   30

/* main section */
Process P_FA()
{
    int a, m1;
    a = sensor_port_a();
    m1 = FA(a);
    send ("P_FD", "P_FA", m1, m1_length, m1_deadline);
}

Process P_FB()
{
    int  b, m2;
    b = sensor_port_b();
    m2 = FB(b);
    send ("P_FD", "P_FB", m2, m2_length, m2_deadline);
}

Process P_FC()
{
    int c, m3;
    c = sensor_port_c();
    m3 = FC(c);
    send ("P_FD", "P_FC", m3, m3_length, m3_deadline);
}
```

Figure A.3: Processes for the three input nodes

```
\* Monitor section *\
Monitor M_FD(m1, m2, m3)
int m1, m2, m3;
{
    int d;
    d = FD(m1, m2, m3);
    output_port_d(d);
}

Process Q_FD()
{
    int m1, m2, m3, d;
    If (EVERY(period_FA,LCM))
       Then receive ("P_FA", , m1, BLK);
       Else receive ("P_FA", , m2, NBLK);
    If (EVERY(period_FB,LCM))
       Then receive ("P_FB", , m2, BLK);
       Else receive ("P_FB", , m2, NBLK);
    If (EVERY(period_FC,LCM))
       Then receive ("P_FC", , m3, BLK);
       Else receive ("P_FC", , m3, NBLK);
    d = M_FD(m1, m2, m3);
}
```

Figure A.4: Monitor and process for the common node

| *primitives* | *descriptions* |
|---|---|
| **send** | the send rendezvous |
| **receive** | the receive rendezvous |
| *attributes* | *descriptions* |
| $L_{S,Q}$ | the worst-case tranmission time for the message betweeen $S$ and $Q$ |
| deadline | the deadline for the message send |
| BLK | the **receive** is a blocking rendezvous |
| NBLK | the **receive** is a non-blocking rendezvous |
| *functions* | *descriptions* |
| input_V() | this is the input function specified at the node $V$. |
| function_V() | this is the function specified at node $V$. |
| output_V() | this is the output function specified at node $V$. |
| EVERY($f$,"$F$") | this function returns a true value if the frequency of invocating "$F$" is equal to the value of $f$. |

Table A.1: **Primitives and functions used in the process and monitor constructions**

# Appendix B

# Pseudo Code and Examples

This appendix gives the outline and pseudo codes used in the task allocation and resource allocation modules. Results generated for other examples are also given.

```
Begin
  Max_node = number of processor nodes available;
  For each task in task_activation_list
     Initialize the execution window,EW for each processor node;
     Compute Tasks Priority,TP;
     Initialize available task set,ATS;
     Initialize latest enable time, LET_1 and LET_2
                for each task in the ATS;
     While (ATS is NOT empty)
        For each (task(j) in ATS)
           If task(j) should be clustered with its BPT, task(i)
              Then Temporarily cluster task(j) with task(i);
                   TPR(j) = MAX[EW(j), LET_2(j)];
              Else TPR(j) = MAX[min_EW, LET_1(j)];
           Update current execution window containing task(j);
        Endfor;
        Select a task in ATS with the smallest TPR;
        If more than one tasks with the same TPR
           Then Select task with the highest task priority;
        Update pseudo window of selected task;
        Update the available task set,ATS;
        Update the LET_1 and LET_2 for each successor task;
     Endwhile;
  Endfor;
End
```

Figure B.1: Pseudo code for the task allocation heuristics

```
Begin
Queue *message_queue;
   /* set up communication_window */
   CW(t) = 0;
   CW_size = group_deadline;

   /* initialize message_queue */
   N = init(&message_queue);
   sort(&message_queue, deadline);

   /* check feasiblity of message (m) */
   For each message(m) in message_queue
      If (lax(m) < CW(t))
         Error("No feasible schedule for messages");
   Endfor;

   /* call time_slot_algo() subroutine */
   Call time_slot_algo(&message_queue);
End;
```

Figure B.2: Outline of the communication time-slot assignment algorithm

```
time_slot_algo(message_queue)
Queue *message_queue;
Begin
   While (message_queue is NOT empty)
      earliest_deadline_queue = get_queue(&message_queue, task_deadline);
      number_of_element = length(&earliest_deadline_queue);
      Switch (number_of_element)
      Case zero: /* queue is empty */
             Return ("End of queue");
      Case one: /* only one message in queue */
         Assign communication time-slot;
         Update message deadline in task_table of (s);
      Case more_than_one: /* more than one message in queue */
          sort(&earliest_deadline_queue, laxity);
          For each message in min_laxity_queue
             message(s) = pop(&earliest_deadline_queue);
             Assign communication time-slot;
             If (message_deadline_new(s) > message_deadline_old(s))
                Then Error("No feasible schedule for messages");
                Else Update message deadline in task table of (s);
          Endfor;
      Endswitch;
   Endwhile;
End;
```

Figure B.3: Pseudo code for the subroutine **time_slot_algo()**

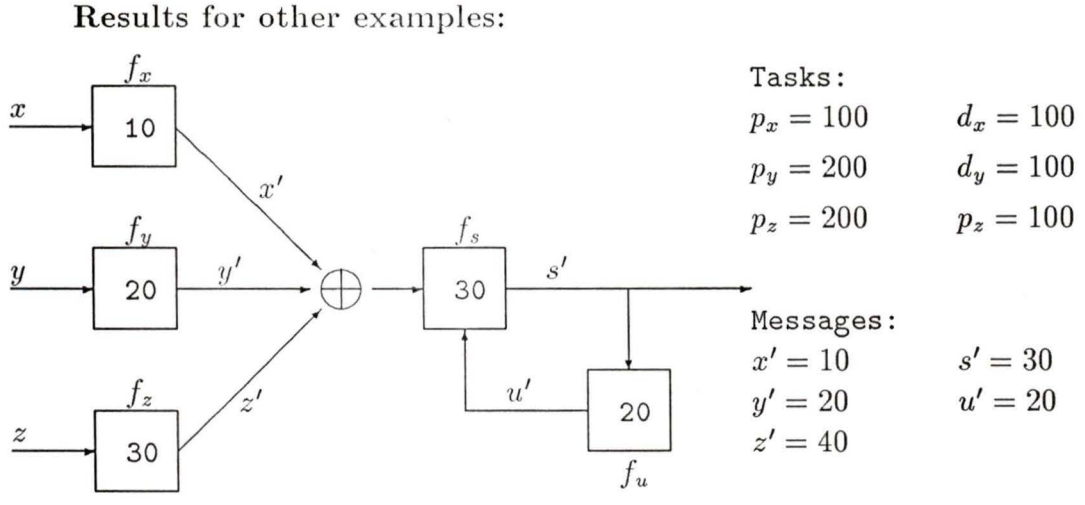



Figure B.4: Example given in Mok's paper[Mok84b]

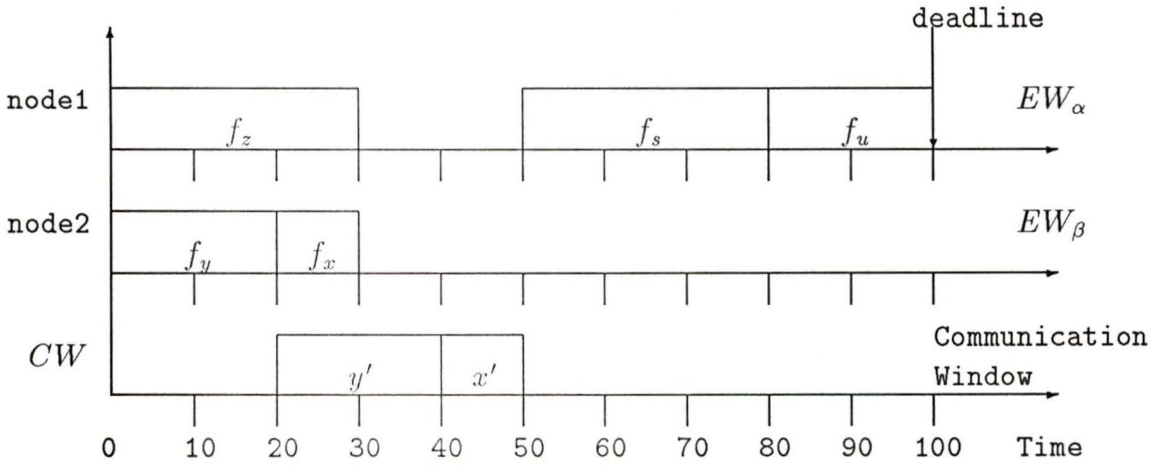



Figure B.5: Assignment of tasks and messages for Mok's example

# Appendix C

# Glossary

This appendix gives a description of the list of abbreviations and functions used in the task and resources allocation modules.

| abbreviations | descriptions |
|---|---|
| $EW$ | current pointer of the execution window |
| $EW\_size$ | execution window's size = group's deadline |
| $min\_EW$ | minimum of all execution windows = MIN(all $EW_s$) |
| $TP$ | task priority indicates the criticalness of executing the task |
| $ATS$ | available task set contains a set of tasks suitable for assignment |
| $EET$ | earliest enable time of a task which is clustered with all its predecessors |
| $LET\_1$ | latest enable time of a task which is not clustered with any of its predecessors |
| $LET\_2$ | latest enable time time of a task which is clustered with at least one of its predecessor |
| $TPR$ | temporary pseudo ready, generated by the heuristic algorithm |
| $BPT$ | binding predecessor task, generated according to the task clustering guidelines |

Table C.1: Abbreviations used in the task allocation heuristics

| abbreviations | descriptions |
|---|---|
| $CW(t)$ | current pointer of the communication window |
| $CW\_size$ | maximum size of the communication window = group deadline |
| $t\_1$ | time-slot's start time |
| $t\_2$ | time-slot's end time |
| $d_m$ | deadline of the message's sending task |
| $L_m$ | message length = worst-case transmission time |
| $D_m$ | message deadline |
| $LAX_m$ | message laxity = $D_m - L_m$ |
| $LS_m$ | latest time to send message |
| $VLS_m$ | virtual latest time to start transmission |

Table C.2: Abbreviations used in the time-slot allocation heuristics

| abbreviations | descriptions |
|---|---|
| $LCM$ | least common multiple of all the periods |
| $PR_s$ | pseudo ready time of the task |
| $PD_s$ | pseudo deadline of the task |
| $RS$ | reserved slot, which is the region where no task should begin execution = $(RS_s, PR_s)$ |
| $RS_s$ | start time of the reserved slot |
| $LS_s$ | latest start time for the first monitor task which accesses the critical section |
| $d$ | deadline of a task |
| $p$ | period of a task |
| $c$ | worst-case computation time of a task |
| $q$ | uninterruptible quantum amount of cpu time |

Table C.3: Abbreviations used in the mutual exclusion algorithm

| function | description |
|---|---|
| $MAX()$ | returns the maximum value of a set of given parameters |
| $MIN()$ | returns the minimum value of a set of given parameters |
| $lax(m)$ | returns the laxity of message(m) = $D_m - L_m$ |
| $init(q)$ | generate a message queue(q) from task_table and returns the size of the queue |
| $sort(q, k)$ | sort queue(q) by ascending key(k) |
| $get\_queue(q, k)$ | return a queue wth the maximum value of key(k) |
| $length(q)$ | return the size of a queue(q) |
| $pop(q)$ | return top element of a queue(q) |
| $MOD(x, y)$ | x modulus y |

Table C.4: Functions used in all the heuristic algorithms

# Vita

Surname: Liew

Given Names: Daniel Leong Wan

Place of Birth: Singapore

Date of Birth: October 21, 1958

Educational Institutions Attended:

| | |
|---|---|
| National University of Singapore | 1978 to 1981 |
| British Computer Society | 1982 to 1986 |
| University of Victoria | 1987 to 1989 |

Degrees Awarded:

| | | |
|---|---|---|
| B.Sc. | 1981 | National University of Singapore, Singapore |
| B.Sc(Hon). | 1986 | British Computer Society, United Kingdom |

Honours and Awards:

Government of Sinagpore Undergraduate Merit Bursary, 1978/81
National University of Singapore Undergraduate Merit Award, 1980/81
Ngee Ann Polytechnic Postgraduate Scholarship, 1987/89

# Partial Copyright License

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

## Methodology For Automating Distributed Hard Real-Time System Development

Author: [redacted]

Daniel L.W. Liew

September 28, 1989

ISBN 0-315-53754-X