SCHEDULING IN TIME TRIGGERED SYSTEMS UNDER MODE CHANGES

By

Ramadass Prabhakar

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

December, 2003

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Benoit Dawant

To my parents, Vishnu, Niranjana and Akka.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my mentor, Dr. Gabor Karsai, who introduced me to the field of Time Triggered Architecture and without whose guidance and advice this thesis would not have been possible.

I also wish to thank Dr. Benoit Dawant, my second advisor for providing invaluable insights.

To Gabor Szokoli, for explaining the nuances of constraint programming. Special acknowledgements are due to Nagarajan Kandasamy and Radhika Tekumalla, for having patiently and painstakingly edited and proofread my drafts.

Many thanks to Beatrice Richardson, members of the ANTS group, Sherif Abdelwahed and everyone at ISIS for their help and support.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

Chapter

I. INTRODUCTION 1

1.1 Task Scheduling 1
1.2 Problem definition 3

II. BACKGROUNDS 5

2.1 Real-time Systems 5
Time-Triggered Architecture 5
2.2 Scheduling 6
2.3 Mode Changes 9
2.4 Constraint Programming 10
Constraint Programming Language 12
Mozart 12
Constraint satisfaction problem 12
2.5 Techniques of Constraint Solving 14
Techniques of Constraint Propagation 17
Search Techniques 17
2.6 Constraint Based Scheduling 20
2.7 Related Work 21
Implementation in Ada 21
Giotto 22
MARS 22

III. MODE CHANGES IN OFF-LINE SCHEDULER 24

3.1 System Model 24
Repetition Window 24
Task 25
Message 26
Latency 27
3.2 Mode Changes 28
Mode Change task and its periodicity 29
3.3 Scheduling Mode changes 30
Offline scheduling algorithm. 31

IV. CASE STUDY 35
4.1 Aircraft Control System 35
4.2 Schedule generation 39
V. CONCLUSION AND FUTURE WORK 46
BIBLIOGRAPHY 47

LIST OF TABLES

Table Page

4.1 Tasks in the system, their duration and period 36

4.2a Tasks executing in Takeoff mode 37

4.2b Messages transmitted in Takeoff mode 37

4.3a Tasks executing in Cruise mode 37

4.3b Messages transmitted in Cruise mode....................38

4.4a Tasks executing in Auto Pilot mode 38

4.4b Messages transmitted in Auto Pilot mode 38

4.5a Tasks executing in Landing mode....................39

4.5b Messages transmitted in Landing mode....................39

LIST OF FIGURES

Figure Page

1.1 Printer mode transitions 2

2.1 Generic scheduler data structures 7

2.2 Domain of variables A and B...11

2.3 Applying constraint A = B 11

2.4 A constraint store and n propagators acting on it. 15

2.5 Constraint store status 16

2.6 Search tree with failure and success states in OZ 19

3.1 Repetition window 25

3.2 Latency between sender and receiver 28

3.3 Mode change task execution across multiple hosts 29

3.4 Case 1: Start time of task A in new mode exactly matches the period.....................32

3.5 Case 2: Start time of A in new mode is lesser than the period 32

3.6 Case 3: Start time of A in new mode is greater than the period 33

3.7 Algorithm for getNextOccurence 33

3.8 Algorithm for getTasksonHost 34

3.8 Algorithm for getEntryPoint...34

4.1 Aircraft control system physical structure 35

4.2 Gantt chart of schedule in Takeoff Mode 40

4.3 Gantt chart of schedule in Cruise Mode 41

4.4 Gantt chart of schedule in Auto Pilot Mode 41

4.5 Gantt chart of schedule in Landing Mode...42

4.6 Gantt chart of schedules on all hosts in all modes 43

4.7 Schedule in Cruise mode corresponding to the 1st MC task (Takeoff mode). 44

4.8 Schedule in Cruise mode corresponding to the 2nd MC task (Takeoff mode). 44

4.9 Schedule in Cruise mode corresponding to the 3rd MC task (Takeoff mode). 45

4.10 Schedule in Cruise mode corresponding to the 4th MC task
(Takeoff mode). 45

CHAPTER I

INTRODUCTION

Task scheduling in complex embedded systems undergoing mode changes, with tasks executing on multiple processors is a very important problem. These tasks communicate with each other through a common medium, the message bus and tasks are executed precisely at pre-determined time intervals.

A typical control system comprises tasks executing on multiple processors, and the ordering and the timing of their execution must be precisely scheduled in order to satisfy non-functional constraints. Scheduling is important, as task execution may depend on results produced by the execution of other tasks and on the number of resources available. Execution of tasks depends upon the timing deadlines imposed by the physical system. Scheduling results in ordering the execution of tasks such that all performance, timing, and serialization requirements are met.

In task scheduling, it has been observed that quite often there is a lot of replication of tasks. Proper allocation of resources for these tasks at the correct time instant can reduce the number of components used in the design of the system, thereby bringing down the cost.

While scheduling tasks, it is more viable to divide the system on the basis of different operational modes than to consider the system as executing in a single mode. Once the operation of the system is divided into modes, schedules can be generated for each of these individual modes. This provides a modular approach to the problem.

1.1 Task Scheduling

Many control systems exhibit mutually exclusive phases of operation called *modes*, where each mode describes an operational phase of the physical system in concern. Mode changes affect the nature of control system in terms of timing requirements and the set of activities to be carried out. For example, a printer can have different modes like “off” mode, “warming up” mode, “ready” mode and “printing” mode. The printer also performs specific tasks during each mode, like “feed paper”, “invert paper”, and “charge drum”. Each mode consists of a number of these tasks. The tasks executing in the different modes maybe the same or the

new mode may execute some additional tasks. It is also possible that some of the currently executing tasks may not execute in the new mode. For instance the "stabilize voltage" task will occur in almost all the modes other than the "off" mode, whereas "feed paper" might occur only in the "printing" mode. Each of these modes may use multiple processors (e.g. actuators, sensors), each of which in turn may execute multiple tasks. A printer also changes modes frequently. For instance from the "off" mode the printer goes into the "warm up" mode. Once it is warmed up, it can start printing. There is a transition from one mode to another and this is called a mode change.

Task scheduling and resource allocation under multiple operating modes of a real-time system is a challenging problem. In real time systems based on time triggered architecture (TTA) [5], mode changes are the only way to bring about a change in the temporal control structure (as the execution order of tasks is pre-computed)[22]. Also, mode changes are an efficient way of providing fault-tolerance in TTA based systems. Recovery from the fault can be handled by switching into a new mode where the necessary steps are taken.

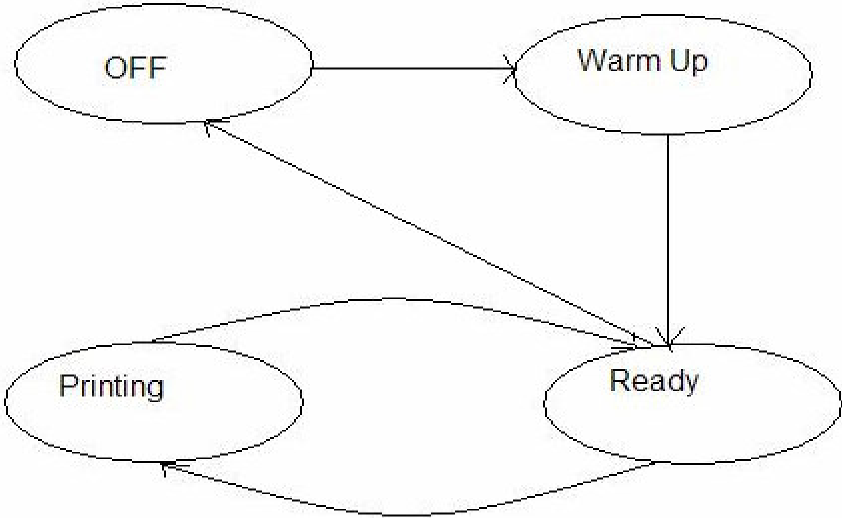

Figure 1.1 Printer mode transitions

1.2 Problem Description

This thesis aims to solve the problem of scheduling in a time triggered system with mode changes using constraint programming. A hard real time system continues executing in its current mode of operation until a change in the environment forces the system to transition to a new mode of operation. This change can be triggered by factors such as component failure in the system, change in operational requirement ("off" in a printer) or passage of time (e.g. in a heating system there could be a condition : shut down heater after 20 minutes of operation).

Requests for a mode change are time-bounded meaning that the transition from one mode to another has to be completed within a specified time interval from the time of request. Transition between modes normally takes some finite non-zero amount of time. Timing constraints are imposed by the system specification and the class of applications that has been targeted in this thesis is clock driven. The execution time of the tasks in these applications is controlled by a global clock. The local clocks in the processors synchronize their time with this global clock within a specified precision providing a uniform time base across all processors [14].

The problem of handling mode changes is solved by creating individual schedules for each mode of operation. The scheduling algorithm [section 3.3] processes the input specifications to set the start times of each task. The inputs to the scheduling algorithm are the set of tasks that need to be executed and the messages to be transmitted between the tasks. The individual schedules generated by the scheduling algorithm are further processed to create an overall schedule which handles each mode transition.

All tasks within a mode are assumed to be periodic, that is they run repeatedly at the rate specified by the task period. Furthermore the possible mode changes are known in advance. A mode change can not occur when a task is still executing. A mode change is possible only if no task is executing at that given instant.

The thesis is organized in the following manner. Chapter 2 provides an introduction to constraint programming and constraint programming techniques which have been widely used to solve similar problems. A brief introduction to time triggered systems, scheduling and scheduling using constraint programming is included. Constraint programming techniques are used to solve the scheduling problem and the language in which the

constraints are encoded is called OZ [6, 11]. This chapter ends with a summary of other work that has been carried out in this field. Chapter 3 provides technical details of the algorithm used to solve the scheduling problem and the design decisions taken in the process. Terms used in context with the thesis are also defined here. Chapter 4 shows an example application of aircraft control and the analysis of the problem and results. Conclusions and future work related to this thesis are provided in the final chapter.

CHAPTER II

BACKGROUNDS

This chapter provides a brief introduction to the various technologies and terminologies used. The solution presented provides a constraint programming approach for solving scheduling problems for a class of real time systems (time triggered systems) undergoing mode changes. A brief introduction to three major concepts: the time triggered systems, scheduling, and the constraint programming framework is provided. The chapter concludes with a summary of related work.

2.1 Real Time Systems

A real time system must react to stimuli from the environment within a deadline dictated by the environment. The correctness of the system depends not only on the computational result but also on the time instant at which the result becomes available [14]. A deadline is defined as a time instant at which a valid result must be available, and a system is said to be a hard real time system if missing a deadline means the system is incorrect.

Real time systems can be broadly classified as event triggered and time triggered [14], where a trigger is an event that causes some action to take place. An event driven system waits for an external signal that indicates the occurrence of an event. Whenever an event occurs, it triggers a corresponding action. This is a reactive system. The problem with this model is that it considers the lack of an event as the absence of a change in the system, whereas the very component which senses the event might be damaged. A failure in this system can also lead to an "alarm shower" [14], i.e., one event rapidly leads to the occurrence of other events causing a chain reaction, making it difficult to pinpoint the cause of the failure.

Time Triggered Architecture

To avoid the aforementioned problem and improve reliability, the time triggered architecture (TTA) was developed as another approach to real time systems at the University of Vienna [14]. The main goal of time triggered architecture is to provide a predictable, reliable, fault tolerant, distributed real time system. In a time triggered system, activities are initiated at

pre-determined points in time. The driving force is a global clock and the periodic clock signal is the only control signal in the system. Each processor in the system synchronizes with the global clock and carries out activities at pre-determined time instants. Aperiodic activities (events) in the system are handled by polling for them periodically. Results (observations) from each activity are time-stamped with the global clock. As a result of time stamping, two observations taken anywhere in the distributed system can maintain their temporal order. Here the system decides when it wants to sample the environment, rather than the environment informing the system of a change. Thus when sampling the environment, the absence of an activity (at the time instant when it is supposed to take place) or an incorrect output value is interpreted as a fault. As a result, if an output is unavailable, fault can be detected immediately. The time instant when an action is to be performed by the system is known *a-priori*, hence its absence can be used to conclude that a fault has occurred. The tasks are executed repeatedly and message transmission is based on a time division multiple access (TDMA) cycle consisting of a sequence of slots corresponding to all the processors in the system. At any given point of time in the TDMA cycle, there can be only one processor which can transmit messages in the slot. The TDMA cycle is designed such that the periodicity of the execution of each task (reading a sensor value or setting a value for the actuator) allows for any fault to be detected. Fault tolerance in a TTA is provided by replication. Node failures in TTA are masked by providing actively replicated nodes (processors). Upon a failure, output from a replicated node is used instead of an output from the failed node. The replicated nodes mask the occurrence of a fault. Redundancy through replication is essential to achieve fault tolerance in the case of a permanent fault. This thesis does not consider the effects of replication in TTA.

2.2 Scheduling

Scheduling problems occur in situations where a set of activities have to be executed by a set of processors, and each activity has a deadline. Scheduling is the process of fixing the start times of the activities to be performed while ensuring that the resources required by these activities are available at that instant. From a resource allocation perspective scheduling is the allocation of resources (processors and the communication bus in our case) to tasks (activities) over a period of time. This defines the start and end of each activity to be

performed. Task scheduling, deals with the question of which task executes next. An example of this is the scheduling of tasks during the construction of a house (e.g. construction of walls has to be completed before laying the roof).

The scheduling methods can be classified on the basis of "when" a schedule is computed as an offline/predictive or online/reactive scheduling [1]. Offline or predictive scheduling as the name suggests is performed before running the system. For this kind of schedule generation, there needs to be sufficient data at the design phase of the scheduler itself. The schedule generated is called predictive as the behavior of the schedule, when it executes is already known.

In online scheduling, the schedule is generated when the system is executing. The nature of the schedule depends on the changes in the system or on the information that is obtained only after the data has been processed at run time. For scheduling mode changes, the offline scheduling technique is used to come up with a predictive schedule. The schedule for the task execution has to be computed before running the system.

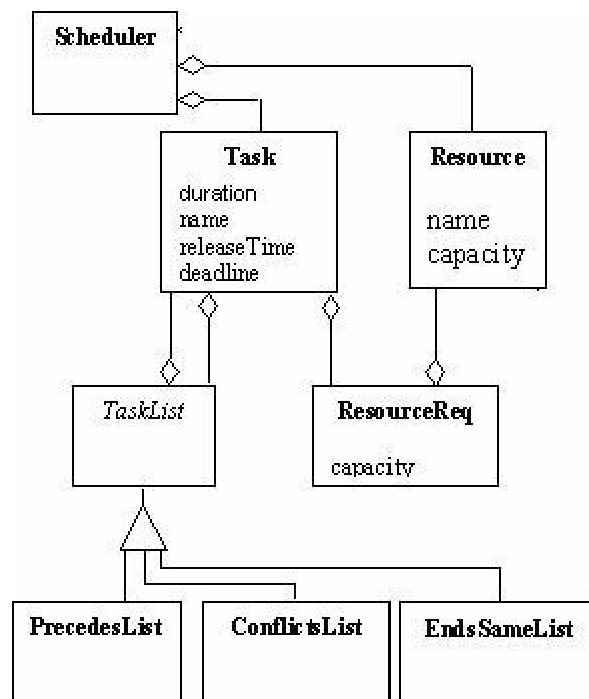

Figure 2.1 Generic scheduler data structures

A schedule consists of a set of tasks and a set of resources required by these tasks. Tasks have the following attributes: name, release time, deadline, periodicity, defined as follows.

- Release time: the earliest point in time when the task can start executing on a processor,
- Deadline: the time by which the task should have completed its execution,
- Periodicity: the rate at which the task should be scheduled to execute.

Execution of a task may be dependent on other tasks. The order of their execution may be enforced by this relationship. Some common relationships are "Precedes", "Conflicts With" "Ends Same". The relation *A precedes B* enforces that the task A has to complete its execution before task B starts. *A Conflicts with B* means that tasks A and B can not execute at the same time. The system may also have a requirement that two tasks complete their execution at the same time ("Ends Same"). In our problem, the precedence relationship between tasks is implicitly enforced by the messages sent between the tasks. If task A sends a message to task B, then task A has to be scheduled to finish execution before task B starts its execution. The periodicity of a task also governs the time period within which the task should be scheduled to execute. Once the start time of one occurrence of a task is fixed, the following occurrences are governed by its periodicity. Tasks have a start time, duration (time taken to complete execution) and a host processor on which it executes.

Scheduling tasks requires start times of the tasks to be fixed. The start times are modeled as finite domain variables (In a finite domain variable, the possible values that the variable can be assigned are restricted to a finite range [2]). Each task has a duration for which it executes. Temporal scheduling constraints (Precedes) can be expressed as arithmetic constraints (constraint is a condition that a correct schedule must satisfy) like

$$t1.Start + t1.Duration < t2.Start$$

This means that a second task (t2) does not begin its execution before the completion of a previously executing task (t1).

Non-overlapping of tasks is described as a disjoint constraint when the task durations are fixed.

$$Disjoint\ (t1.Start, t1.Duration, t2.Start, t2.Duration)$$

means that $t1.Start + t1.Duration \leq t2.Start + t2.Duration$ or

$t2.Start + t2.Duration \leq t1.Start + t1.Duration$.

Tasks whose start time maybe different but have to complete their executions at the same time (endsSame) can be specified as

$$t1.Start + t1.Duration = t2.Start + t2.Duration$$

More scheduling specific constraints are described in chapter 3.

2.3 Mode Changes

Most scheduling techniques avoid solving mode change problems by assuming that all the tasks occur in a single schedule (a single mode). If all the tasks execute in a single mode, then the information about which tasks are coherent is lost. Tasks are said to be coherent if they contribute towards a single goal. For instance when an aircraft is landing, the tasks "open wheels", "landing gear down" are coherent as they contribute towards a single goal (landing the aircraft). Having multiple modes each consisting of coherent tasks helps design the system using the "divide and conquer" methodology. It is also easier to test for different modes individually than for the entire system as a single mode. The thesis solves the mode change problem by generating schedules for each mode individually and then these schedules are processed to handle mode changes.

There have been numerous methods for implementing mode changes in various real time systems [8, 18, 19]. Most of the approaches have focused on event based systems. There have been few applications which have solved the mode change problem for time driven real time system [18, 19]. Schedules for time triggered systems are computed at pre-runtime, hence the system can not adapt to changes at run time. Mode changes are important for TTA based systems as this is the way by which change in the system conditions at run time can be handled. Some of the methods solve the problem at the hardware level [22]. Other solutions do not consider the time taken to switch between modes [18, 21]. In control systems when there is a switch between modes, there is some delay which is involved in the initialization of the new mode. Also, some solutions interrupt the task executing at the time of a mode change request [18]. In this thesis, the mode change problem is solved using constraint programming techniques and by the use of an offline scheduler for TTA [4]. This method provides a predictable way of handling mode changes as the time instants when a mode change can occur in the system are fixed, and also the task schedule that will be executed after a mode

change is pre-computed. Details of the algorithm and the technique used to solve this problem are presented in Chapter 3.

2.4 Constraint Programming

Schedules in this thesis have been generated using constraint programming techniques, and this section provides an introduction to constraint programming.

In the words of Eugene C. Freuder –

> "*Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.*" [24]

Constraint programming is a problem solving paradigm which establishes the distinction between precise definition of constraints that define the problem and the algorithms and heuristics that enable selection and cancellation of decisions to solve the problem [1].

In traditional programming languages, "if A > B then S1" is a conditional statement, which means "if the value of A is greater than B then execute the statement S1". In constraint programming, constraints apart from being used as condition statements are also used to prune the domain of values of variables. Pruning is carried out by removing the values from the domain which do not satisfy a constraint.

Let A and B be two variables, initially A can take any value from the set {2, 3, 4, 6, 9, 11} and B can take any value from {1, 2, 5, 6, 7, 10} Fig 2.2 and 2.3. Let us say that there is a new constraint A = B added to the system. Using constraint programming techniques (discussed in following sections) all the values from the value set which do not satisfy this relationship are removed. This would return the intersection of sets A and B i.e. {2, 6}. Thus the constraint reduces the possible values that A and B can be assigned to.

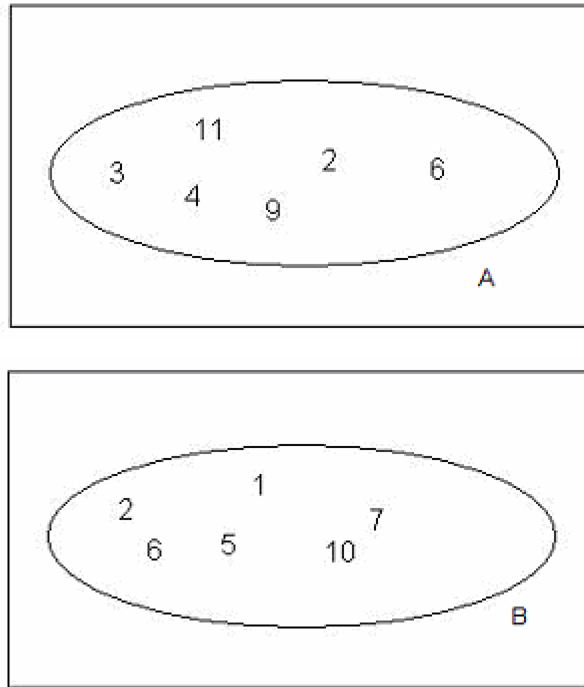

Figure 2.2 domain of variables A and B

Applying constraint A = B, values that A and B can take reduce to the region depicted by dashed lines.

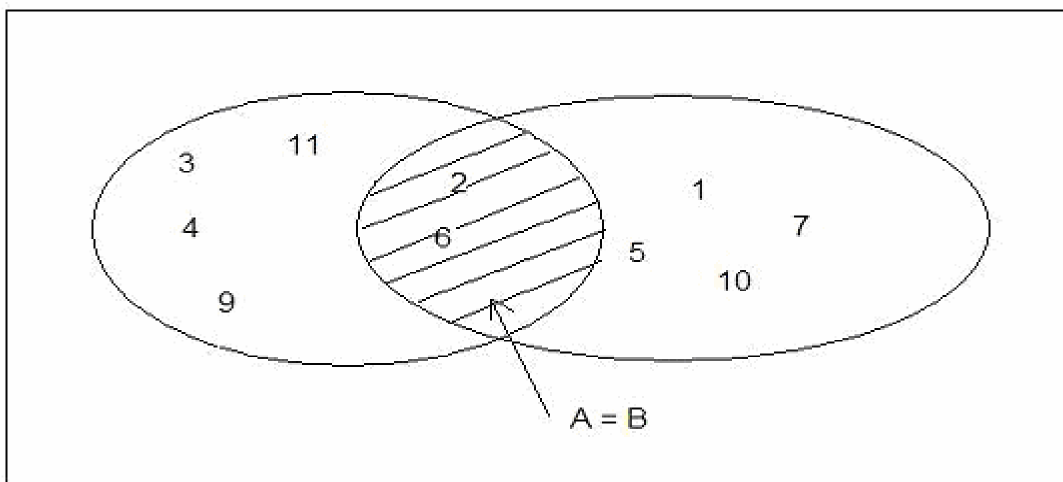

Figure 2.3 Applying constraint A = B

Constraint Programming Languages

For numerous applications, modeling the relationships between objects and searching for objects that satisfy them are very important. A language which lets the programmer define the relationships between objects and leave the maintenance of these relationships to the underlying implementation is called a constraint programming language. These relationships are called constraints [2]. For example, consider a system with three variables A, B and C, where the relationship between these variables is given by the constraint A = B+C. This constraint controls the values that these variables can be assigned. A constraint is said to be violated if the values assigned to the variable do not satisfy the given constraint. For instance A= 5 B= 2 C= 1 is an incorrect assignment because this violates the rule A = B + C.

Mozart

Mozart is an advanced development platform for intelligent, distributed applications and is based on the Oz language. Oz is a high-level programming language that is designed for modern advanced, concurrent, intelligent, networked, soft real-time, parallel, interactive and pro-active applications [11]. Oz is described as the first high-level constraint language which allows the developer to program search [6]. It allows the user to program their own search mechanisms in addition to the already provided strategies like branch and bound or depth-first search [7]. This thesis uses the constraint programming features of Oz.

Various features of Oz, like propagation and distribution, are described in the following section [section 2.5]. Apart from the generic, built-in techniques, Oz also allows us to customize the propagation and distribution algorithms. Utilizing this feature it is possible to create new heuristics based strategies to solve the constraint satisfaction problem.

Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is a constraint problem in which the goal is to find a consistent assignment of values to variables [20]. CSP is formally defined by a finite set of variables, the set of values that these variables can take and the constraints which restrict the combination of values that these variables can take. Solving a CSP is equivalent to finding an assignment for all variables such that all the constraints are satisfied. Revisiting the above mentioned example, the domain (set of values that the variable can take) of A, B and C is

[1..5], implying that the variables A,B and C can take any value from 1 to 5. Suppose the constraints are A=B+C and A=5. The set of variables, their value domain and the constraints define the CSP completely. One solution set to the problem is the assignment A = 5, B = 3, C = 2.

To find a consistent assignment for the variables, the constraints are repeatedly applied to the domain of the variables to remove values which do not satisfy the constraints. The process of repeated application of constraints is called propagation, and the domain is said to be "pruned". After one round of propagation, the value of A is fixed by the second constraint A = 5. Thus the domain of A is pruned to 5.

The process of pruning continues till the value of all the variables has been fixed.

The main task of constraint propagation is to remove inconsistent values from the domain of the variables [3]. In most cases, propagation alone does not necessarily yield a solution as explained in section 2.5.

The advantage that constraint programming offers over traditional programming languages is that traditional languages provide very little support for specifying relationships between objects and entities defined by the programmer and the programmer has to explicitly maintain these relationships, whereas in a constraint programming language, this is handled efficiently by the constraints themselves [2] .

For example, to calculate simple interest, the relationship between the principal P,
rate R, time T and the interest I is given by I = P * R * T /100.

Given any three of the unknowns the fourth variable can be calculated. In traditional programming languages the programmer will have to maintain the relationship explicitly. He has to encode statements to find the value. Thus, if he has to calculate the principal he has to use the equation

P = I * 100 / (R *T).

But if he has to calculate the time period given the other three values, he has to encode

T = P * R / (I * 100)

The programmer has to encode different statements to get the values of each variable, whereas in constraint programming the value of the variables in the relationship is maintained and can be obtained from the relationship itself.

Thus, only a single encoding I = P * R * T /100 is needed and the value for any single unknown can be directly obtained from this relationship.

Constraint programming techniques have been successful as it is easy to model complex problems and the modeling is done mainly by specifying the constraints between the variables. As a result of this, changing the model can be done by adding and/or removing constraints. The use of constraints is becoming popular in the fields of artificial intelligence, databases, combinatorial optimization, etc. [10]

The drawbacks of constraint programming are that controlling search is still an active research area and there are few generic, high performance techniques available to facilitate search. Another drawback is the lack of efficient debuggers. Most constraint programming languages do not provide a built-in debug facility.

2.5 Techniques of Constraint Solving

There are numerous methods of solving constraints. Techniques commonly used are simplification, optimization, bounds propagation techniques and integer programming techniques [2]. Simplification is used to make the implicit information apparent. It is a process of replacing a constraint by another constraint which has a simpler form [2]. Optimization techniques are used when there is a need to find the best possible solution. There are also consistency based techniques like arc and node consistency developed by the Artificial Intelligence community [2]. These have been widely used to solve scheduling and routing problems [23].

Two techniques that are used explicitly by the OZ programming language are

- Constraint propagation
- Constraint distribution.

Constraint propagation in Oz uses concurrently working propagators which constantly add information about the variables to a constraint store [11]. A propagator is a concurrent computational agent that propagates the information represented by a constraint thereby narrowing the domain of its variables. For example if the domain of a variable A is [1..15], a constraint A <8, when propagated, would reduce the domain of A to [1..7]. Two propagators sharing a variable can communicate with each other through the above mentioned store. The store contains information about the values of variables as a conjunction of the basic

constraints [10]. These concurrently working propagators reduce the domain of the variables with the addition of information to the store. In the above example if another constraint A>3 is added, then another propagator will further reduce domain of A to [3..8].

Distribution is a method which splits the problem into complementary cases when the propagation can not proceed any further [9]. Distribution is explained in a later section on search.

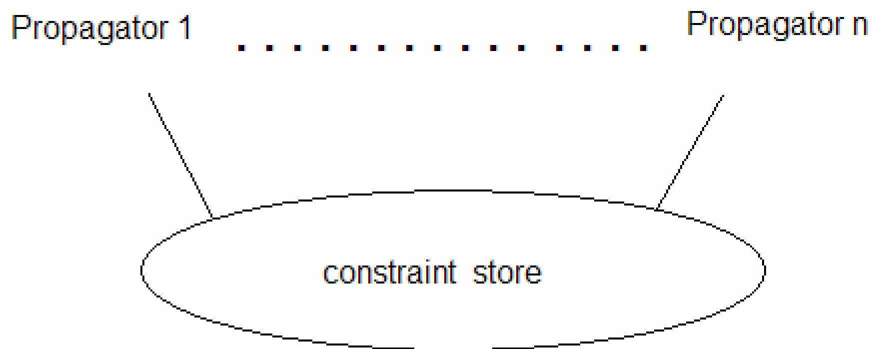

Figure 2.4 A constraint store and n propagators acting on it.

Described below is an example for propagation.

Let us take two constraints

$X + Y = 9$ and

$2X + 4Y = 24$

The domains of X and Y are [0..9]

The store initially contains the above domain information.

The first propagator doesn't do anything, but the second propagator reduces the domain to new values of X and Y such that

X = [0..8] and Y =[2..6]

This reduction can be explained as follows. If X = 9, then there is no value of Y that can satisfy the propagator. Similarly for Y = [0, 1, 7, 8, 9] there are no values of X in the domain that satisfy the above constraint. Hence they are removed from the domain.

Now applying the first propagator again

X = [3..7] and Y = [2..6]

The second propagator makes it

X = [4..6] and Y = [3..4]

Now the first one becomes active and the domains are further curtailed

X = [5..6] and Y = [3..4]

Now the second propagator fixes the value of the variables as

X = 6 and Y = 3

Figure 2.4 shows the contents of the store at each stage. The status of the store changes as shown in the figure. The lines from the constraints on the left to the constraint show which constraint is pruning the store and the final result is X: 6 and Y: 3.

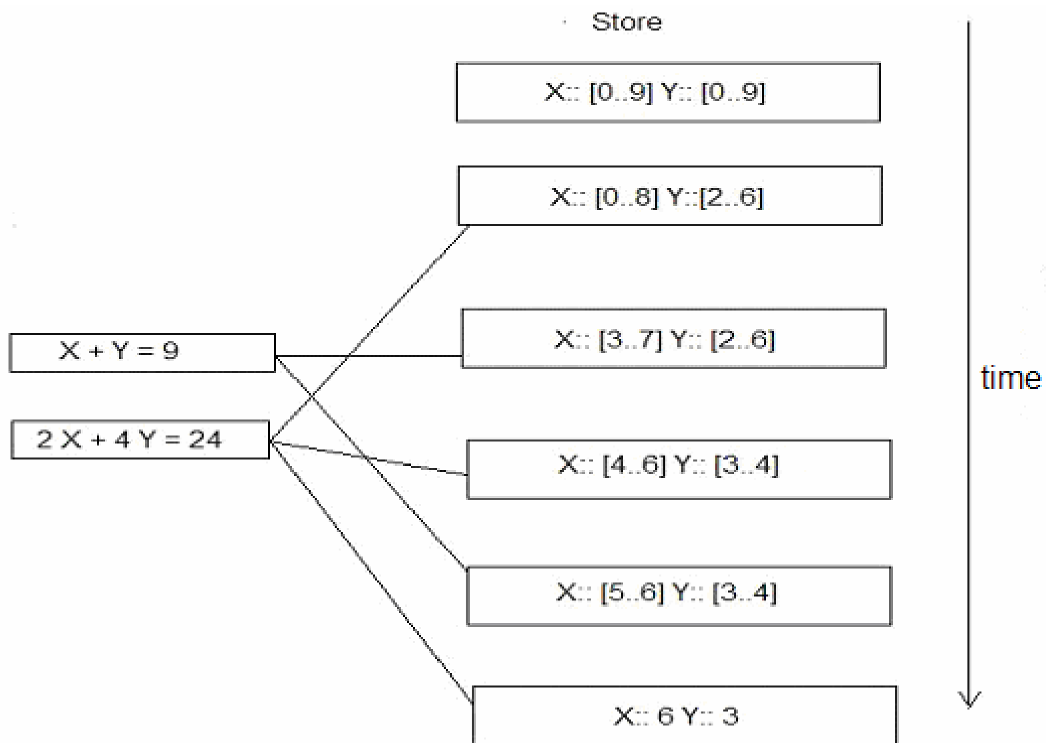

Figure 2.5 Constraint store status

Techniques of Constraint Propagation

Propagators can be implemented in two ways, either as Interval Propagation or as Domain Propagation.

In Interval Propagation, the propagator only narrows the domain of the variable. So it only provides us with a change in either the Upper Bound and/or the Lower Bound.

Domain Propagation removes as many values from the Domain of the variables as possible.

Let us take the propagator

A * B = 8

A = [0..10]

B = [0..10]

With interval propagation, the values of A and B can be

A = [1..8]

B = [1..8]

On applying domain propagation

A = [1, 2, 4, 8]

B = [1, 2, 4, 8]

In general, domain propagation gives a much finer result but its computation is more expensive and hence interval propagation is used more often. The default propagator in Oz uses interval propagation.

Search Techniques

Constraint propagation alone is not enough to solve problems: it is not a complete solution procedure. This is true in cases where the space becomes stable, but the problem is neither solved nor has failed and no further propagation is possible with the current constraints. A space is said to be stable when constraints working on it can no longer prune the domains. Then it becomes essential to start search. Search proceeds by looking for a solution in a search tree, which is a representation of the search process. The root of the search tree represents the state of the search process at start. Leaf of a search tree denotes termination with a solution (success) or with an inconsistent solution (failure). Intermediate nodes in the tree represent the intermediate states in the search process (see Figure 2.6). The designer can control the shape as well as the exploration of the search tree [10]. Distribution (also called

branching or labeling) is one method by which search can be carried out. Distributing a space over a constraint C creates two spaces, one by adding the constraint C to the space and another by adding ¬C (NOT C). This introduces a branch point in the search tree. Propagation can take place either in the space created by C or in the space created by ¬C, based on the search algorithm. Propagation continues till the space becomes stable or a solution or failure is reached. If a solution or failure is not reached, then a not yet determined variable 'x' and a value "L" from the domain of 'x' is chosen and the space is distributed over the constraint x = L. This repetitive procedure leads to the exploration of the search tree. In terms of Oz each node corresponds to a space and a leaf can indicate either a failed space or a success space.

There are a few standard possibilities for choosing a value for the variable 'x' when distribution takes place in OZ [11].

- Distribute x = L, such that L is the least possible value for x.
- Distribute x = L, such that L is the largest possible value for x.
- Distribute x = L, such that L is the median of all possible values of x.
- Distribute x ≤ L, such that L is the median of all possible values of x.

The process of selecting which variable to distribute on next is called a distribution strategy. A *naïve* distribution strategy would select the leftmost undetermined variable (a variable whose value is not fixed) from the list of variables for distribution. A "first fail" strategy will select the leftmost undetermined variable in the list of variables which have the smallest domain in the constraint store. In most cases the first fail strategy yields a smaller search tree. The order in which a search tree is explored can have an impact on the memory resources needed to find a solution. In Oz the search tree is explored by default in a depth first manner and on distributing with constraint C, the space obtained with C is explored first before ¬C is explored.

Almost all constraint languages provide a minimal set of search strategies. Many of these languages do not allow the users to implement their own search mechanism, but in Oz search can be controlled by specifying user defined distribution and search strategies thus making the language very powerful. This thesis uses the default built-in strategies for search: the search mechanism used is depth first search and the distribution strategy is the naïve strategy.

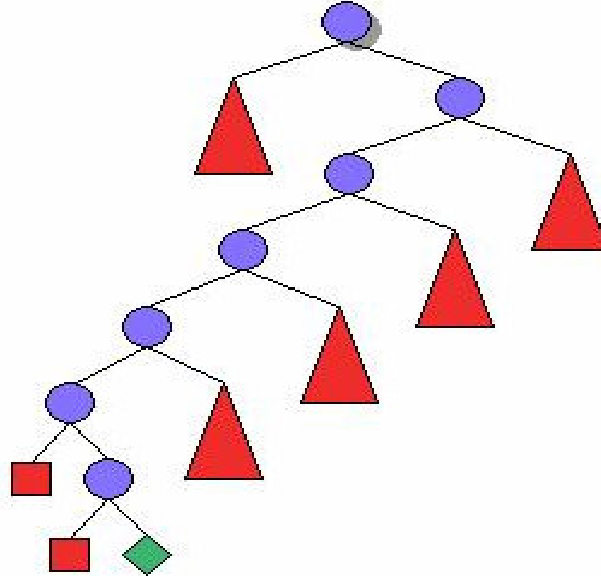

Figure 2.6: search tree with failure and success states in OZ

The above figure shows a sample search tree. The squares denote failed nodes, the diamond denotes a success state and the triangle denotes a collection of failure nodes. Each circle is a distribution point.

The search process can be divided into refinement based methods and repair based methods [12]. Refinement based methods are the most widely used, where each of the variables is assigned a value until a complete solution is found or a constraint is violated. The method works by selecting an unassigned variable from the set of variables. A value is assigned to the variable from its domain. If a constraint is violated, search backtracks to the last distribution point and tries an alternate path. This process repeats for all unassigned variables. Repair based methods start with a complete assignment and on encountering a violation, the assignment is repaired. This repair is done by assigning different values to one or more variables (which had an inconsistent assignment initially) till a solution is reached. One heuristic that can be applied is to select the variables that are known to violate the maximum number of constraints. The repair based technique is not complete and do not necessarily find an assignment that satisfies all the constraints. The search process in Oz is based on refinement based techniques.

Some other well known search algorithms are limited discrepancy search (LDS) [13], branch and bound, best solution search [2]. An Oz implementation of LDS was not available, and

hence not used in this thesis. Both branch and bound and best solution search optimize the output, and search for all possible solutions. This search is rather time consuming and hence not used in our solution.

2.6 Constraint Based Scheduling

Over the years various approaches have been devised to solve practical scheduling problems. One of the recent techniques has been the development of constraint-solving frameworks and search procedures to find a solution. Various scheduling-specific constraints and domains have been developed recently. Variable domains specific to scheduling include *interval domains* where each value in the domain is an interval. The corresponding scheduling specific constraint is *interval constraint* [12]. An example of scheduling specific propagation techniques is *resource timetables*, which maintains a timetable with required and available capacity at any given time for each resource [17].

Constraint based scheduling techniques are being used in a number of industrial applications, for e.g. for paper path control in Xerox copiers [12].

2.7 Related Work

As mentioned earlier, there have been a few attempts to solve the mode change problem. This section provides a brief description of three of the major approaches undertaken.

Implementation in Ada

One of the earlier works in the field of mode changes was carried out by Jorge Real and Andy Wellings [8]. They suggested an implementation of mode changes with respect to resource management in Ada. The main goal was to ensure that there was no inconsistent use of shared resources during mode transitions.

Resources are represented by protected objects in Ada. Each object in the system is assigned a ceiling priority using the priority ceiling protocol (PCP) [16]. The priority ceiling protocol is used to schedule a set of periodic tasks that have exclusive access to common resources protected by semaphores. Let A, B and C be three tasks with A having the highest priority and C the lowest priority. Tasks A and C require exclusive access to a common resource. Considering a scenario where C has access to the shared object and then the task A is

requested. Task A also requires the same object, but it can not start execution as it is being held by C. In the meanwhile task B is requested and it pre empts C and starts executing. Now as a result C is delayed, resulting in A the highest priority critical task being delayed. This is called priority inversion. To solve this, the “priority inheritance protocol” was designed, where in the lower priority task’s priority is temporarily increased to that of the higher priority task. This protocol could lead to chained blocking and deadlocks [14]. To overcome this problem the PCP was developed. The ceiling priority is the highest priority that can be allotted to tasks using this object. Objects can be assigned different ceiling priorities in different modes. Difference in the ceiling priority during transitions could lead to an inconsistent use in the following manner. During a mode change a task still executing in the old mode could try to access an object with a priority higher than the new ceiling priority thus violating PCP. As a result the shared resource can be accessed by objects which do not have sufficient privileges. This indicates that priority ceiling values of the object have been changed when the object is being accessed. This leads to an inconsistent state in the system where the priority of an object is higher than its ceiling priority.

Two methods have been suggested to handle this inconsistency. The first method is to abort currently executing tasks on a mode change or to allow the currently executing task to complete before doing the mode change. The other method activates the relatively important tasks in the target mode before the tasks in the old mode complete as long as they do not conflict. The handling of mode changes and the execution of tasks is handled by an object with the highest priority called “Mode_Manager” [8].

The drawback of either approach is that, they solve the mode change problem for a single processor system and not for a distributed system. This overall approach to solve the mode change problem allows a task which is executing to be interrupted during it execution, and this can be hazardous in a hard real time system. Also it does not consider the delay that takes place when switching between modes. In control systems mode transitions always take some time. The advantage of this implementation in Ada is that it provides a predictable and offline solution to mode change problem.

Giotto

Giotto is a time triggered language for embedded control. It is based on the principle that time-triggered task invocations plus time-triggered mode switches can form the abstract essence of programming control systems [21]. It aims to provide flexibility in choosing control platforms by separating the platform independent concerns from the platform dependent ones. Platform independent issues are timing and application functionality. Scheduling and communication are platform dependant issues. Giotto provides an abstract programmers model decoupling software design from its implementation.

A Giotto program does not specify when, where and how tasks are scheduled [18].

Giotto primarily comprises of periodic task invocations and time-triggered mode switches. A Giotto program consists of a set of modes and each mode determines the tasks and mode switches in it. Each task executes at a given frequency as long as the mode is unchanged and each mode has mode switches which are evaluated periodically at a given frequency. If the mode switch condition evaluates to true, then a mode change takes place.

Tasks communicate with each other through ports. A port is a persistent typed variable with a unique location in a globally shared namespace. There are three kinds of ports- sensor ports, actuator ports and task ports. Data is communicated through task ports. Each task has a set of input ports and a set of output ports and output ports can be shared with other tasks as long as they are not invoked in the same mode. Each task has a driver which is a function that converts the values from sensor and mode ports to values for input ports of the target mode. During a mode change the mode driver passes values of the ports from one mode to another. This is carried out with the output port of the current mode as the source and the output ports of the target mode as the destination. If a task is running at the time of a mode switch, then the next mode must contain this task. Giotto assumes that mode changes are instantaneous, i.e. time taken to achieve mode change is zero.

MARS

Work under the MARS project was carried out at the Technical University of Vienna. They consider a mode to represent an operational phase which is performed by a single schedule [19]. A transition between modes represents a change in the control information of the system and the mode change is an instantaneous event. The prerequisites on the system are

that it should have a deterministic temporal behavior and transitions should have timing constraints associated with them. Dependencies between tasks and their execution orders are expressed by precedence constraints. These constraints are represented by directed, acyclic graphs called precedence graphs [19]. Tasks are represented by nodes and precedence relations by the arcs. The maximal response time of the graph is the time between the start of the first task and the completion of the last task. Information about the modes in which these tasks are applicable is maintained in the edges (arcs). When switching from one mode to another, transition precedence graphs are made use of, which are executed only once during the initiation of a mode change. On the other hand, the precedence graphs for the individual modes run periodically. The transition precedence graph helps in the smooth transition between two modes. The transition precedence graph is designed by the designer and this brings up three possibilities.

- The transition graph is empty implying an abrupt change in the modes
- The transition graph is similar to the graph of the old mode, indicating that all currently executing tasks have to be completed.
- Transition graph which contains some parts of the old mode and parts of the new mode, indicating that it contains important activities from the old mode as well as the activities from the new mode needed to initialize the transition.

Changing from one mode to another could involve multiple transition precedence schedules. All mode changes are designed by specifying a state machine and the user is forced to specify the precedence graphs with their timing constraints. This state machine also provides information for testing possible transitions.

In control systems, it is possible that while switching modes, the timing constraints of certain task executions may need to be maintained across mode changes. The MARS approach does not provide information as to how the period of a task can be maintained across mode changes. Also tasks are allowed to be interrupted abruptly during their execution, if a mode change request is made.

CHAPTER III

MODE CHANGES IN AN OFF-LINE SCHEDULER

This chapter provides the details of the design for the approach used to solve the scheduling problem. In the first section of the chapter, the system model is described. This is followed by a detailed explanation of the method followed to solve the mode change problem and the chapter ends with a description of the algorithm that was designed.

3.1 Modeling Assumptions

This section describes the system modeling assumptions. The task and message models used in this thesis and the constraints imposed on them are explained.

Repetition Window

It is an interval of time in which a collection of tasks is executed, in a specific, fixed sequence, and which is repeated while the system is active. The end of one instance of the repetition window is followed by the beginning of the next instance. All tasks (a task can be repeated within the repetition window) must be executed at least once within the repetition window. A sample repetition window is shown in the figure (3.1). Task T1 is executed twice within one repetition window. Cycle time is the length of the repetition window and is calculated as the least common multiple (LCM) of the period of all tasks in the system.

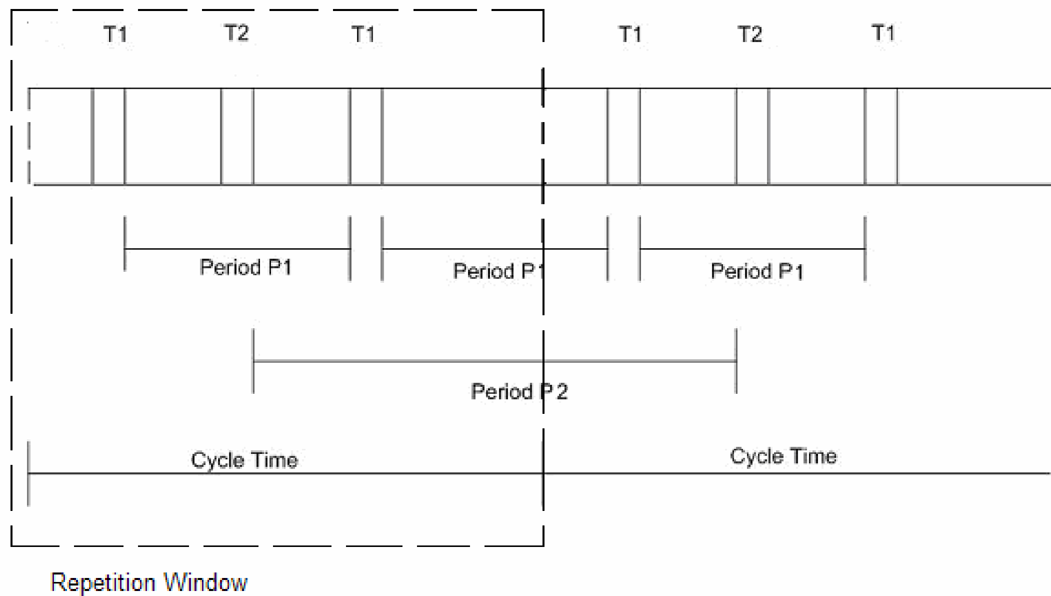

Figure 3.1 Repetition window

Task

A task corresponds to the operation executed on a processor and is defined by the tuple

$T_i = \{ N, H, D, P\}$

N- Name of the task

H- Host on which the task executes

D- Duration of the task

P – Period of the task

A host is the processor on which the task executes, duration is the time taken to complete execution, period represents the frequency with which the task must execute, i.e., the task period is the 10 ms, then its execution must start exactly every 10 ms.

Each task has a period with which it executes and it is possible that in one repetition window there are multiple occurrences of the task. Only one task can execute on a host at any given time.

The constraint that encodes the relationship between tasks is shown below.

The start times of the multiple occurrences of the task are given by

$t_i.start = t_{i-1}.start + T_i.period$, for $i \geq 2$

The start time of the current instance of a task differs from the start time of the previous instance of the same task exactly by the task's period. This holds for all instances of the task other than the first instance i.e. t_1. t_i represents the current instance of the task, t_i.start denotes the start time of the ith task t_i and T_i.period is the period of execution of the task T_i.

The completion time "t_i.complete" of the task is the sum of its start time and its duration.

$t_i.complete = t_i.start + T_i.duration$

Execution of a task should complete within the cycle time.

$t_i.complete <= CT$

There can be multiple tasks executing on the same processor and the order of tasks has to be fixed. Also there are constraints that tasks can start execution only after the completion of the other task, for example, task A completes execution either before task B begins or vice versa. Thus no two tasks on the same processor can have overlapping executions. This is expressed by the constraint:

For tasks P and Q

$P_i.complete \leq Q_j.start$ or $Q_j.complete \leq P_i.start$

whenever P_i and Q_j ($P_i \neq Q_j$) execute on the same host.

This implies that either task P_i finishes its execution before task Q_j commences its execution or task Q_j completes its execution before task P_i starts its execution.

Message

Execution of tasks may depend upon the information produced by other tasks. Tasks communicate with each other via messages. The system comprises of only one common message bus and all tasks communicate using this message bus. Messages are represented by the tuple

$M_i = \{N, T_S, R_i, D\}$

N- Name of the message

T_S- The sender task

R_i- The receiver tasks

D- Duration of the message

The sender is the task which puts the message on the message bus. There can be more than one task receiving the message and once the message is put on the message bus, the bus starts

transmitting the message immediately. The system requires some time to finish transmitting the message after the message is placed on the bus and during this time, the message bus is not available for transmitting other messages. The period of a message is the same as that of the sender, and a message can have exactly one sender.

Start time of the message is constrained by

$m_i.start = m_{i-1}.start + T_S.period$, for $i \geq 2$

The start time of the current instance of a message differs from the start time of the previous instance of the message by the period of sender. $m_i.start$ denotes the start time of ith instance of the message m. T_S is the task that transmits the message and $T_S.period$ denotes the period of the sender.

The completion time "$m_i.complete$" of the message is the sum of its start time and its duration.

$m_i.complete = m_i.start + M_i.duration$

The execution of a message should complete within the cycle time.

$m_i.complete \leq CT$

As at any given time there can be only one message on the message bus, the order of transmission of messages has to be fixed. For two messages m_i and n_j

$m_i.complete \leq n_j.start$ ot $n_j.complete \leq m_i.start$

where m_i and n_j are inter-processor messages and $m_i \neq n_j$.

There is also a relation between the message and the execution of the task. Messages are transmitted only after the sender finishes its execution

$t_S.complete \leq m_i.start$

The sender t_S completes its execution "$t_S.complete$" before the transmission of the message m_i begins.

Latency

There can be relative timing constraints between tasks allocated to different hosts. Latency is an upper bound on the time that may pass between the start of the execution of the sender of a message and the completion of the execution of the receiver with the message transmission taking place in between [6]. This guarantees that the information processing is completed

within the specified time. Latency between the sender P1 and receiver Q1 is shown in the figure.

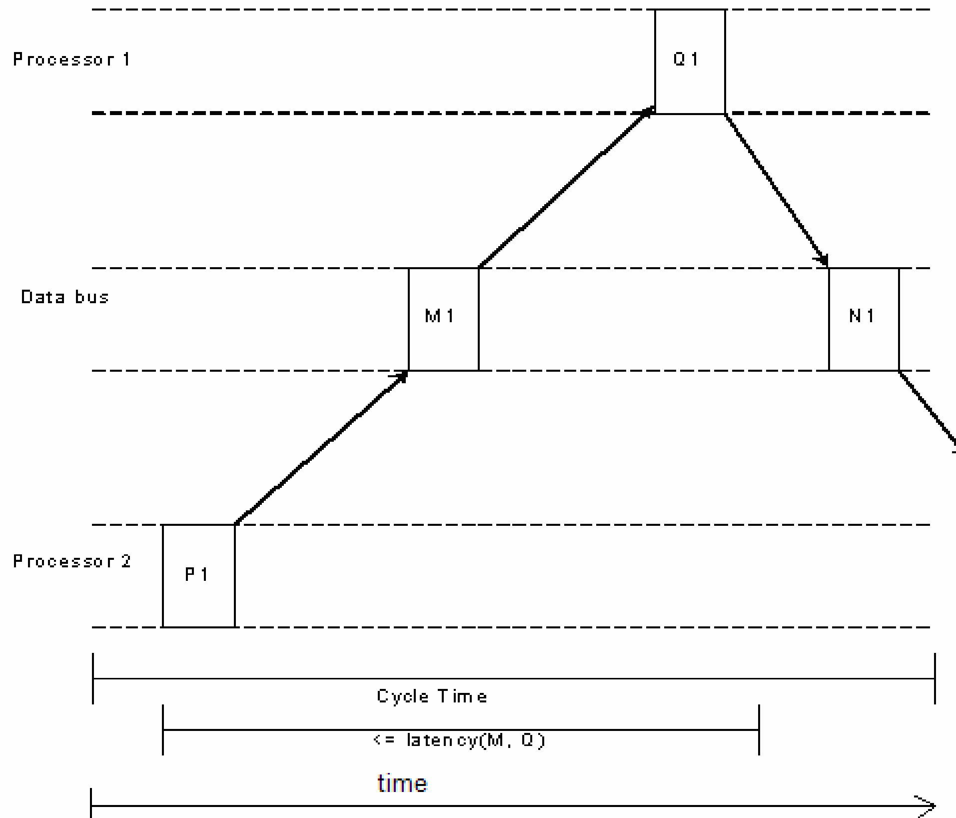

Figure 3.2 Latency between sender and receiver.

3.2 Mode Changes

A system can have multiple modes where the actions performed can be different in each of them. A mode is entirely defined by the set of tasks that have to be executed in it. Each mode also consists of the messages that are passed among these tasks. In our solution, an additional task called the *mode change* task is introduced into each mode. A mode change can occur only when the mode change task is executing in the current mode, but it is not necessary for a mode change to happen for each occurrence of a mode change task. The notion of a mode change here is a global one, i.e. the entire system moves from one mode to another: the execution of the mode change is synchronized across the hosts i.e., the mode change task

executes on all hosts at the same time within the cycle (see figure 3.3). This allows avoiding the interruption of the executing tasks.

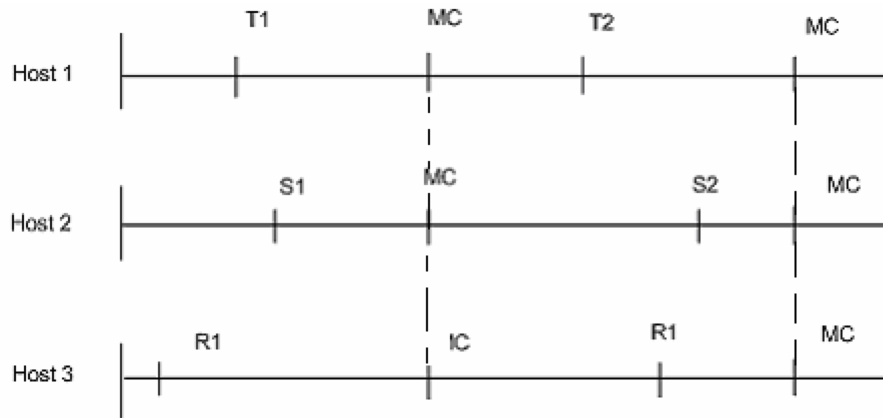

Figure 3.3 Mode change task execution across multiple hosts

The designer can specify the period of execution of the MC task as an input to the scheduler. The system is designed such that a request for a mode change is accommodated only when the system is executing a mode change task. Whenever a MC task is encountered, the system checks to see if a mode change request has been made. The default period of the MC task is the period of the most frequently occurring task in that mode and its duration is one time unit. The responsiveness of the system to a mode change request is dependant on the period of the MC task. Responsiveness to a mode change request is the maximum delay from the time of request to the time when the request is handled. This implies that handling any mode change request will start within a time period equal to that of the mode change task. In control systems when there is a change from one mode to another, there is some time taken to initialize the system for the new mode. This time delay is called the mode delay. The execution in the new mode can not start before this delay.

Mode Change Task and its Periodicity

To provide predictability in the system, the points in time in the system where a mode change request can be satisfied are fixed by mode change tasks. These mode change tasks are added to input as additional tasks to be scheduled by the scheduler. The period of the mode change task can be specified in two ways.

The first method is to match the period of the MC with the period of task with the least period, in other words with the same frequency as that of the task which occurs most often. The reason for this is that the system should be able to handle a mode change request before the next occurrence of any task. This guarantees that between two consecutive occurrences of any task, a mode change request will be handled. This design strategy could make it hard for the scheduler to come up with a schedule even though one existed before the addition of a mode change task. This implementation is suitable for systems where handling mode changes is very critical.

The second method allows the designer to specify the periodicity with which he wishes to check for a mode change request. If the designer has a requirement which stipulates that a request has to be serviced within a specified time period, he can specify this as an input. The most suitable period is to take the median of the periods of all tasks. In the system there maybe a large number of tasks with a given period, then the period of the mode change task should be set to this value as this guarantees that for the majority of tasks a mode change request will be serviced before its next occurrence. The designer can use this as a guideline to decide the period of the mode change task. This methodology requires some preprocessing of the input data. Also it is possible that due to the introduction of mode change tasks it is not possible to generate a schedule, but this is accepted as a necessary risk to provide for predictable handling of mode changes.

3.3 Scheduling Mode Changes

The process of scheduling under mode changes is divided into three parts

1. Preprocess the input to decide the period of the mode change task.
2. Use an offline scheduling algorithm to calculate task schedules for each mode independently.
3. Process the schedules for all the modes and create a final mode change schedule. The user can specify a particular task such that it maintains its timing across mode changes.

Offline Scheduling Algorithm

This algorithm implements an offline scheduling algorithm for time triggered real time systems. A potentially indefinite periodic processing has to be mapped onto a single repetition window [4]. The real time system has the following characteristics:

- Multiple processors: Each processor has tasks executing on it, and each task executes on only one processor.
- Single data bus: Message communication between processors is carried out through a single data bus. This communication is synchronous with the period of the executing tasks.
- Tasks are non pre-emptive and periodic.
- Each message has only one sender but can have multiple receivers.

The start times for all the messages and the tasks are fixed by the scheduler within the repetition window. The input to the offline scheduler is a list of tasks, list of messages, list of processors (hosts) and the latency constraints. The structures for tasks and messages have been defined in section 3.1. The input data is checked to ensure that all tasks have a period larger than its duration time. The cycle time is calculated as the LCM of the period of all tasks. Next, the constraints explained in section 3.1 are asserted. The constraints are propagated, and a naïve distribution strategy is used to search in a depth first manner. The output of the scheduler is composed of two schedules: one for tasks and the other for messages. The task schedule contains records of tasks along with the start times of their execution and the message schedule contains records of messages with the start times of their execution.

The offline scheduler, with the introduction of the mode change tasks is run for each different mode, generating independent schedules for each mode. These individual schedules have to be processed to generate a final schedule which accommodates the mode changes across the multiple modes. The mode change schedule is generated by calculating the entry point in the target mode schedule for each mode change task. The algorithm to implement this is shown in figure 3.9. The entry point holds for all the processors, i.e. in the target mode all the hosts enter the mode at the same time instant on a mode change. One of the requirements can be to maintain the timing of a specific, user-selected task (“selectedTask”).

Figures 3.4 - 3.6 show the possible scenarios when switching modes. Let task A and MD denote the selectedTask and the mode delay respectively. A^t is the first instance of task A in the target mode and there can be multiple instances of task A in the target mode.

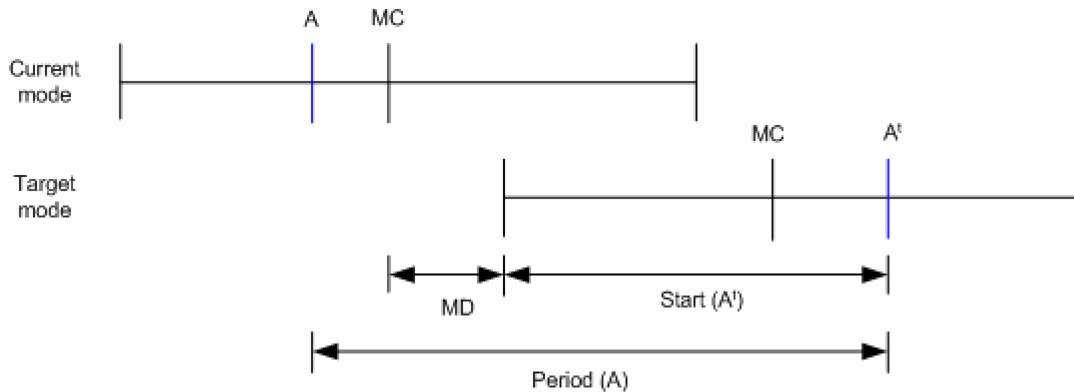

Figure 3.4 Case 1: Start time of task A in new mode exactly matches the period.

Case 1: if A.startTime + A.period = MC.startTime + MD + A^t.startTime

That is, the period of selectedTask A will be maintained, if execution in the target mode starts from the beginning of the cycle. Thus upon a mode change, the new mode should start executing from the beginning of its repetition window.

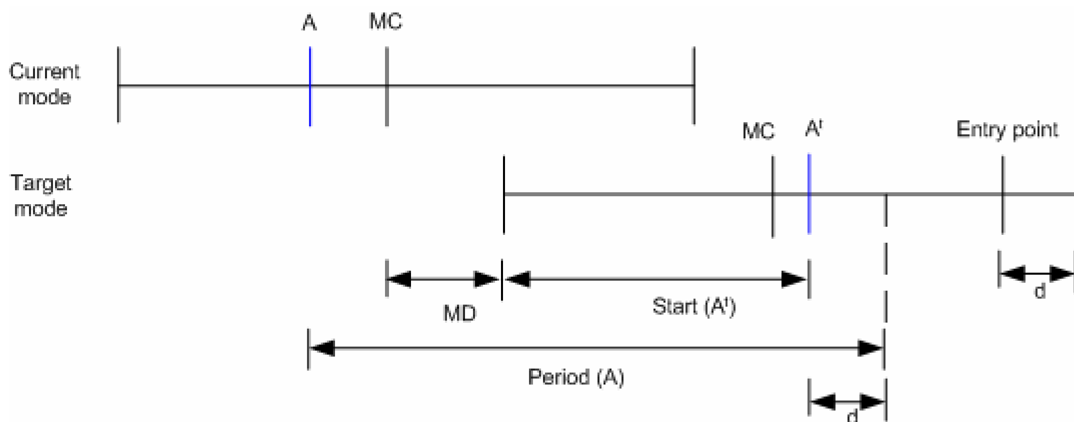

Figure 3.5 Case 2: Start time of A in new mode is lesser than the period.

Case 2: if A.startTime + A.period > MC.startTime + MD + A^t.startTime

That is, to maintain the period of A, the execution of the schedule should start *d* time units before the end of the repetition window, where,

d = (A.startTime + A.period) - (MC.startTime + MD + A^t.startTime) . The entry point in the repetition window is show in the figure (3.5).

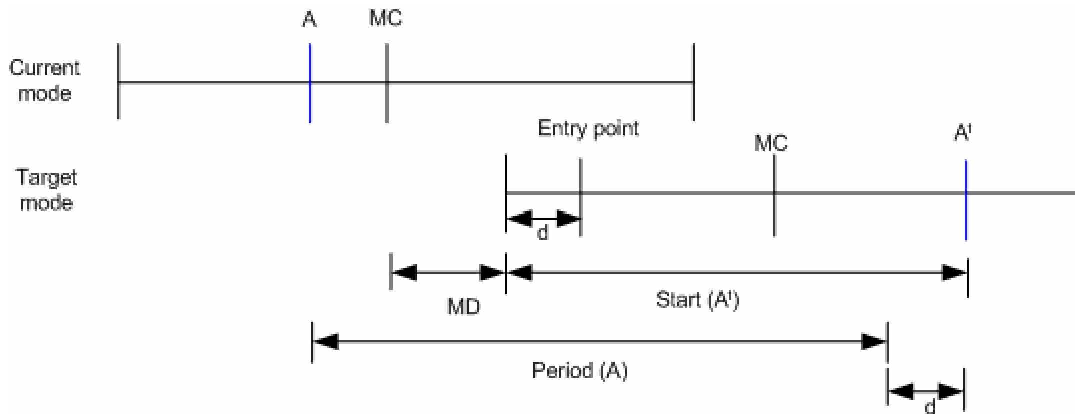

Figure 3.6 Case 3: Start time of A in new mode is greater than the period.

Case 3: if A.startTime + A.period < MC.startTime + MD + A^t.startTime

Then, the period of A will be maintained, if the execution of the schedule starts *d* time units from the start of the repetition window where,

d = (MC.startTime + MD + A^t.startTime) - (A.startTime + A.period). This is denoted by "entry point" in the figure (3.6).

```
// Returns the start time of the last execution of the task in the given list.
Input  : taskList, selectedTask
Output: nextOccurence

for each task in taskList do
  if task.name = selectedTask then
     nextOccurence := task.startTime + task.period
  end if
end for
```

Figure 3.7 getNextOccurence : to get the next occurence time of a given task

```
// Returns a list of tasks on a particular host in the given mode
Input  : Mode, hostId
Output: tasksonHost
for each task in Mode do
   if task.hostId = hostId then
      tasksonHost.add (task)
   end if
end for
```

Figure 3.8 getTasksonHost : returns tasks on a given host

```
//This method calculates the entry point for each MC task in the target mode.
Input  : Problem, selectedTask, modeDelay
Output: entryPointinTarget

for each mode in Problem do
   taskList := getTasksonHost (mode, hostId)
   for each task in taskList do
      if task.name = selectedTask.name then
         StartTime := task.startTime
         taskPresent := true
      end if
      if task.name = MC then
         if StartTime > 0 then
            nextOccurence := task.startTime + task.period
         else
            nextOccurence := getNextOccurence()
         end if
         if (nextOccurence – (task.startTime + modeDelay)) < 0 then
            nextOccurence += selectedTask.period
         end if
         targetModeTasks := getTasksonHost(nextMode, hostId)
         taskFoundinTarget := false
         for each targetTask in targetModeTasks do
            if (targetTask.name = selectedTask) and (taskFoundinTarget = false) then
               startTimeinTarget := targetTask.startTime
               taskFoundinTarget := true
            end if
         end for
         entryPointinTarget := (nextOccurence – (task.startTime + startTimeinTarget
                                                                    + modeDelay)
      end if
   end for
end for
```

Figure 3.9 getEntryPoint : returns the entry point in the target mode

CHAPTER IV

CASE STUDY

This chapter provides an example of an aircraft control system. The different modes of operation in the aircraft control system are described. The input tasks as given to the scheduler and the schedules generated in each mode are represented as Gantt charts, and the output from the mode change schedule is explained.

4.1 Aircraft Control System

The aircraft control system consists of four major operational units. The inertial navigation unit (INU) measures linear and angular acceleration, a global positioning system (GPS) for measuring position, an air data measurement system (ADMS) for measuring air pressure and a pilot control system (PCS). These units communicate with each other by means of a common message bus. The physical system structure is show in figure 4.1.

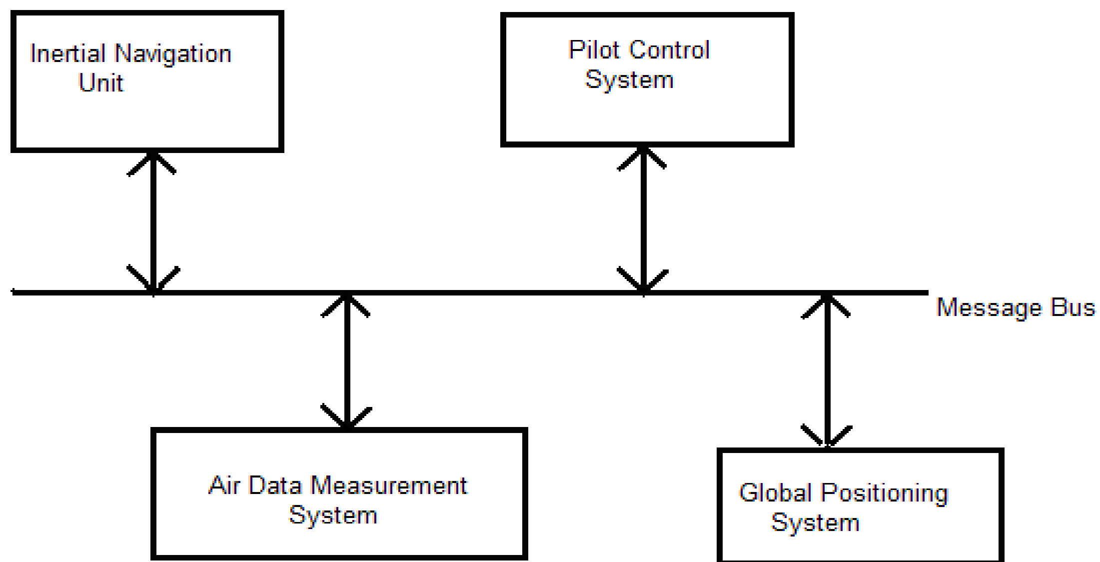

Figure 4.1 Aircraft control system physical structure

The tasks that can execute in each of these operational units are shown in the table 4.1. The table provides information about the period of execution of these tasks, their duration and the

name with which they are referred to in future tables. The aircraft control system operates in four different modes namely, Takeoff, Cruise, Auto Pilot and Landing. Initially the system is in the Takeoff mode from which, the system switches to the Cruise mode. From the Cruise mode the system moves into the Auto Pilot mode before entering the Landing mode. The tasks executing in each of these modes are shown by tables 4.2 (a) to 4.5 (a) and the messages transmitted between these tasks are shown by tables 4.2 (b) to 4.5 (b).

Table 4.1 Tasks in the system, their duration and period.

Host Name	Task Name	Symbolic Name	Duration	Period
INU	Gyros	A	10	100
	Accelerometers	B	5	50
	Aileron 1	C	5	50
	Pitch Control	A1	5	100
	Gyro-Astro compass	B1	4	25
	Angular Accelerometers	C1	5	50
	Reflectometer	C2	5	25
PCS	Rudder pedal	D	10	100
	Throttle control	E	10	50
	Nose landing gear	D1	5	100
	Nose wheel steering	E1	10	50
	Main landing gear	E2	5	50
ADMS	Air data transducer 1	F	10	50
	Air data transducer 2	F1	10	50
	Barometric altitude	F2	5	25
	Thermal protection system	F3	10	100
GPS	Altimeter	G	10	50
	Lateral control	H	4	25
	Tailplane/elevator	G1	5	50
	Vertical control	G2	5	100

Table 4.2a Tasks executing in Takeoff mode

<table>
<tr><td>Host ID</td><td colspan="3">Host 1 (INU)</td><td colspan="3">Host 2 (PCS)</td><td colspan="3">Host 3 (ADMS)</td><td colspan="3">Host 4 (GPS)</td></tr>
<tr><td rowspan="5">Task Details (Name, Duration, Period)</td><td>A</td><td>10</td><td>100</td><td rowspan="2">D</td><td rowspan="2">10</td><td rowspan="2">100</td><td rowspan="5">F</td><td rowspan="5">10</td><td rowspan="5">50</td><td rowspan="2">G</td><td rowspan="2">10</td><td rowspan="2">50</td></tr>
<tr><td>B</td><td>5</td><td>50</td></tr>
<tr><td>C</td><td>5</td><td>50</td><td rowspan="3">E</td><td rowspan="3">10</td><td rowspan="3">50</td><td rowspan="3">H</td><td rowspan="3">4</td><td rowspan="3">25</td></tr>
<tr><td>A1</td><td>5</td><td>100</td></tr>
<tr><td>B1</td><td>4</td><td>25</td></tr>
</table>

Table 4.2b Messages transmitted in Takeoff mode

Message Name	Sender	Receivers	Duration
M1	A	D	2
M2	D	A	2
M3	B	E, F	2
M4	E	B, C	2
M5	C	E, F	2
M6	F	B, C	2
M7	G	F	2
M8	H	B1	2
M9	G	C	2

Table 4.3a Tasks executing in cruise mode

<table>
<tr><td>Host ID</td><td colspan="3">Host 1 (INU)</td><td colspan="3">Host 2 (PCS)</td><td colspan="3">Host 3 (ADMS)</td><td colspan="3">Host 4 (GPS)</td></tr>
<tr><td rowspan="5">Task Details (Name, Duration, Period)</td><td>A</td><td>10</td><td>100</td><td rowspan="2">D</td><td rowspan="2">10</td><td rowspan="2">100</td><td rowspan="2">F</td><td rowspan="2">10</td><td rowspan="2">50</td><td rowspan="5">G</td><td rowspan="5">10</td><td rowspan="5">50</td></tr>
<tr><td>B</td><td>5</td><td>50</td></tr>
<tr><td>C</td><td>5</td><td>50</td><td rowspan="2">E</td><td rowspan="2">10</td><td rowspan="2">50</td><td rowspan="2">F1</td><td rowspan="2">10</td><td rowspan="2">50</td></tr>
<tr><td>C1</td><td>5</td><td>50</td></tr>
<tr><td></td><td></td><td></td><td>E1</td><td>10</td><td>50</td><td>F2</td><td>5</td><td>25</td></tr>
</table>

Table 4.3b Messages transmitted in cruise mode

Message Name	Sender	Receivers	Duration
M1	A	D	2
M2	D	A	2
M3	B	E, F	2
M4	E	B, C	2
M5	C	E, F	2
M6	F	B, C	2
M7	G	F	2
M8	F1	E	2
M9	F	D	2
M10	D	F1	2
M11	E	F	2

Table 4.4a Tasks executing in auto pilot mode

<table>
<tr><th>Host ID</th><th colspan="3">Host 1 (INU)</th><th colspan="3">Host 2 (PCS)</th><th colspan="3">Host 3 (ADMS)</th><th colspan="3">Host 4 (GPS)</th></tr>
<tr><td rowspan="5">Task Details (Name, Duration, Period)</td><td>A</td><td>10</td><td>100</td><td>D</td><td>10</td><td>100</td><td rowspan="2">F</td><td rowspan="2">10</td><td rowspan="2">50</td><td rowspan="2">G</td><td rowspan="2">10</td><td rowspan="2">50</td></tr>
<tr><td>B</td><td>5</td><td>50</td><td>E</td><td>10</td><td>50</td></tr>
<tr><td>A1</td><td>5</td><td>100</td><td rowspan="2">E1</td><td rowspan="2">10</td><td rowspan="2">50</td><td rowspan="2">F2</td><td rowspan="2">5</td><td rowspan="2">25</td><td rowspan="2">H</td><td rowspan="2">4</td><td rowspan="2">25</td></tr>
<tr><td>C1</td><td>5</td><td>50</td></tr>
<tr><td>C2</td><td>5</td><td>25</td><td>E2</td><td>5</td><td>50</td><td>F3</td><td>10</td><td>100</td><td>G1</td><td>5</td><td>50</td></tr>
</table>

Table 4.4b Messages transmitted in auto pilot mode

Message Name	Sender	Receivers	Duration
M1	A	D	2
M2	D	A	2
M3	C1	E, F	2
M4	E	C1, C	2
M5	C	E, F	2
M6	F	C1, C	2
M7	G	F	2
M8	H	F2	2
M9	G1	F	2
M10	F2	C2	2
M11	F3	A	2
M12	A1	F3	2
M13	E1	F	2
M14	E2	C1	2
M15	E2	F	2

Table 4.5a Tasks executing in landing mode

<table>
<tr><td>Host ID</td><td colspan="3">Host 1 (INU)</td><td colspan="3">Host 2 (PCS)</td><td colspan="3">Host 3 (ADMS)</td><td colspan="3">Host 4 (GPS)</td></tr>
<tr><td rowspan="7">Task Details (Name, Duration, Period)</td><td>A</td><td>10</td><td>100</td><td>D</td><td>10</td><td>100</td><td rowspan="4">F</td><td rowspan="4">10</td><td rowspan="4">50</td><td rowspan="2">G</td><td rowspan="2">10</td><td rowspan="2">50</td></tr>
<tr><td rowspan="2">B</td><td rowspan="2">5</td><td rowspan="2">50</td><td rowspan="3">D1</td><td rowspan="3">5</td><td rowspan="3">100</td></tr>
<tr><td rowspan="2">H</td><td rowspan="2">4</td><td rowspan="2">25</td></tr>
<tr><td>C</td><td>5</td><td>50</td></tr>
<tr><td>A1</td><td>5</td><td>100</td><td rowspan="3">E1</td><td rowspan="3">10</td><td rowspan="3">50</td><td rowspan="3">F2</td><td rowspan="3">5</td><td rowspan="3">25</td><td rowspan="2">G1</td><td rowspan="2">5</td><td rowspan="2">50</td></tr>
<tr><td>B1</td><td>5</td><td>25</td></tr>
<tr><td>C2</td><td>5</td><td>25</td><td>G2</td><td>5</td><td>100</td></tr>
</table>

Table 4.5b Messages transmitted in landing mode

Message Name	Sender	Receivers	Duration
M1	A	D	2
M2	D	A	2
M3	B	E1, F	2
M4	E1	B, C	2
M5	C	E1, F	2
M6	F	B, C	2
M7	G	F	2
M8	H	F2	2
M9	G1	F	2
M10	F2	C2	2
M11	B1	F2	2
M12	C2	H	2
M13	E1	F	2
M14	C2	F2	2
M15	D1	G2	2
M16	G2	D	2
M17	B1	H	2

4.2 Schedule Generation

Schedules are generated for each mode individually using the offline scheduling algorithm explained in chapter 3. The input to the scheduler is the data shown in tables 4.2 to 4.5. The corresponding schedule generated in each mode is shown by a Gantt chart. The Gantt chart

shows a single repetition window. This schedule is repeatedly executed till a mode change is requested. The figure shows the names of the tasks executing at a given time instant: each rectangular colored block in the figure represents a task and the duration of the task is equal to the width of the block. The horizontal axis represents time and the vertical axis represents the different processors on which tasks execute.

In figure 4.2 task B executes on host 1, for time duration of 5 milliseconds (ms), followed by task C for 5 ms and B1 for 4 ms. The mode change task MC executes at time instant 15 ms for duration of 1 ms. Similarly on host 2, task E executes for 10 ms followed by MC task at time instant 15. Task B on host 1, task E on host 2 and task H on host 4 execute at the same time instant.

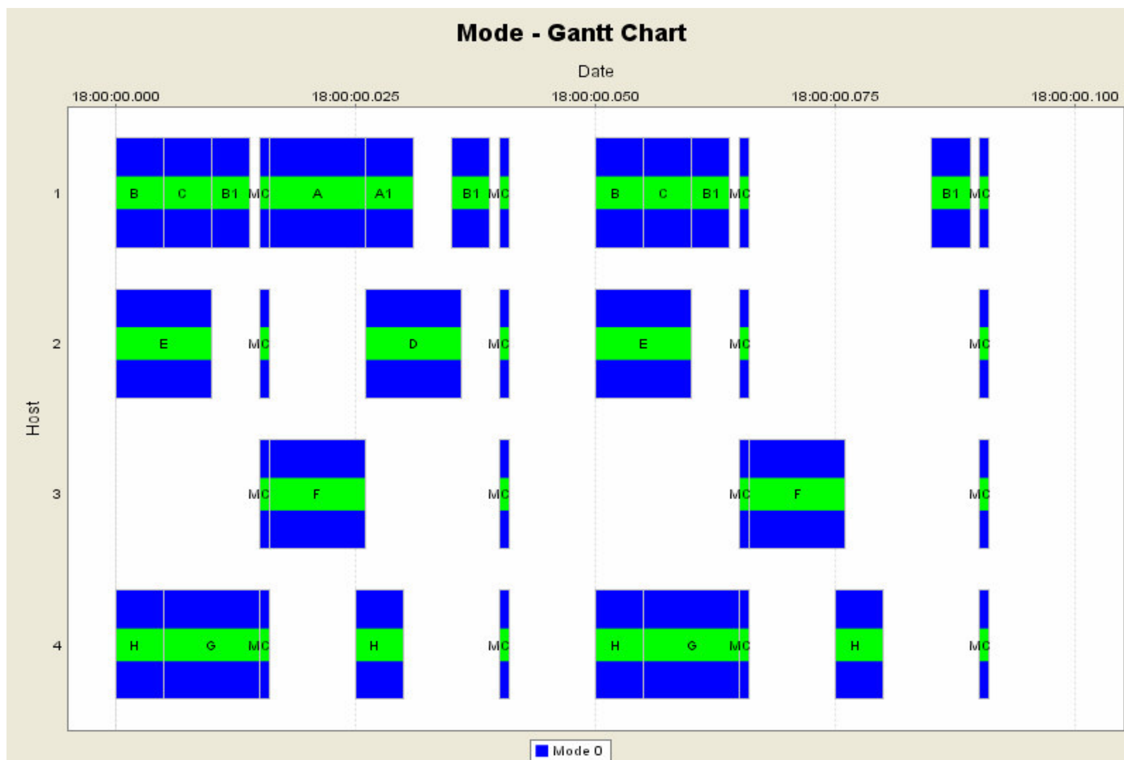

Figure 4.2 Gantt chart of schedule in Takeoff Mode

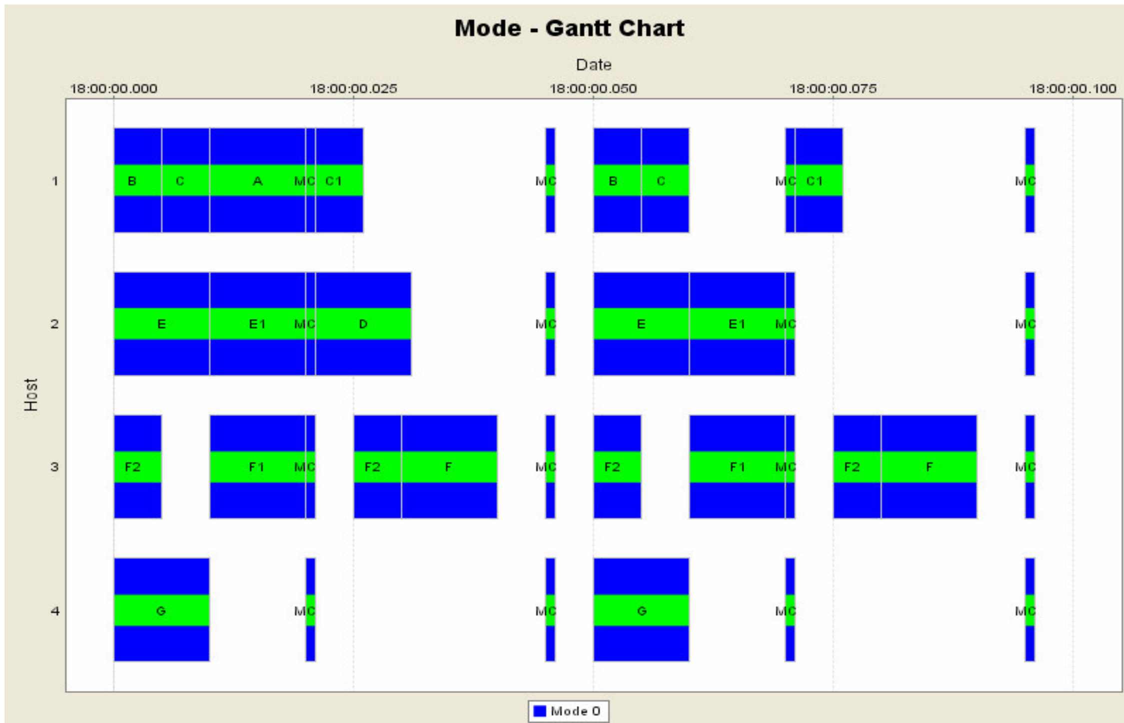

Figure 4.3 Gantt chart of schedule in Cruise Mode

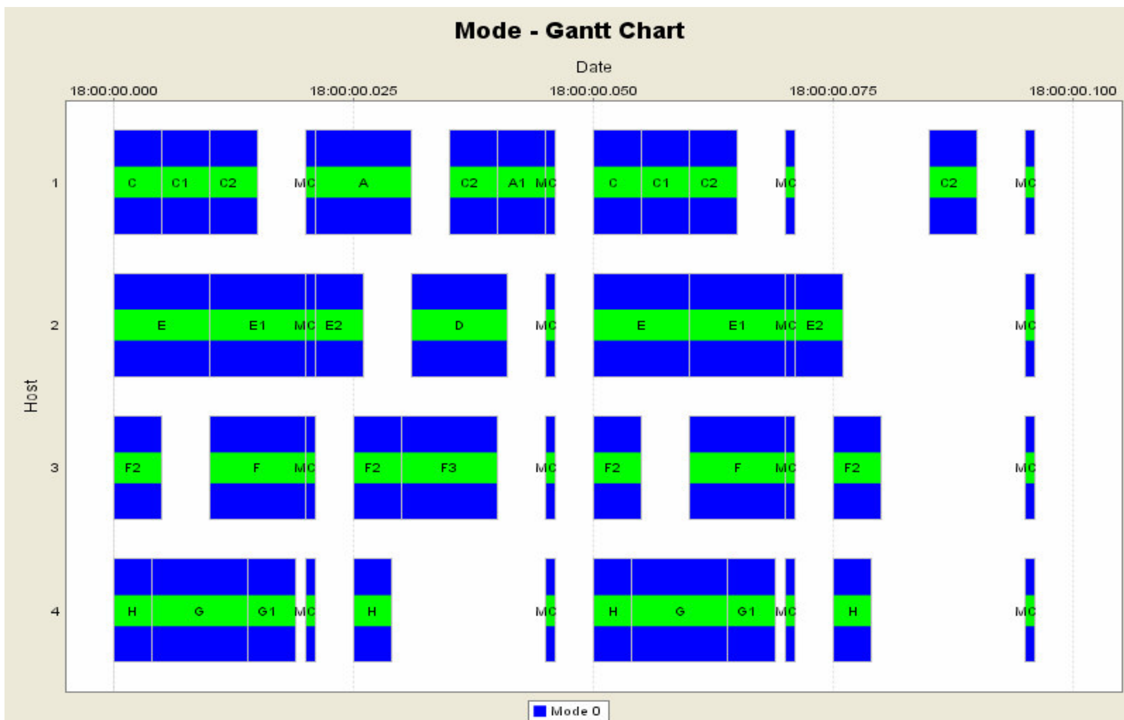

Figure 4.4 Gantt chart of schedule in Auto Pilot Mode

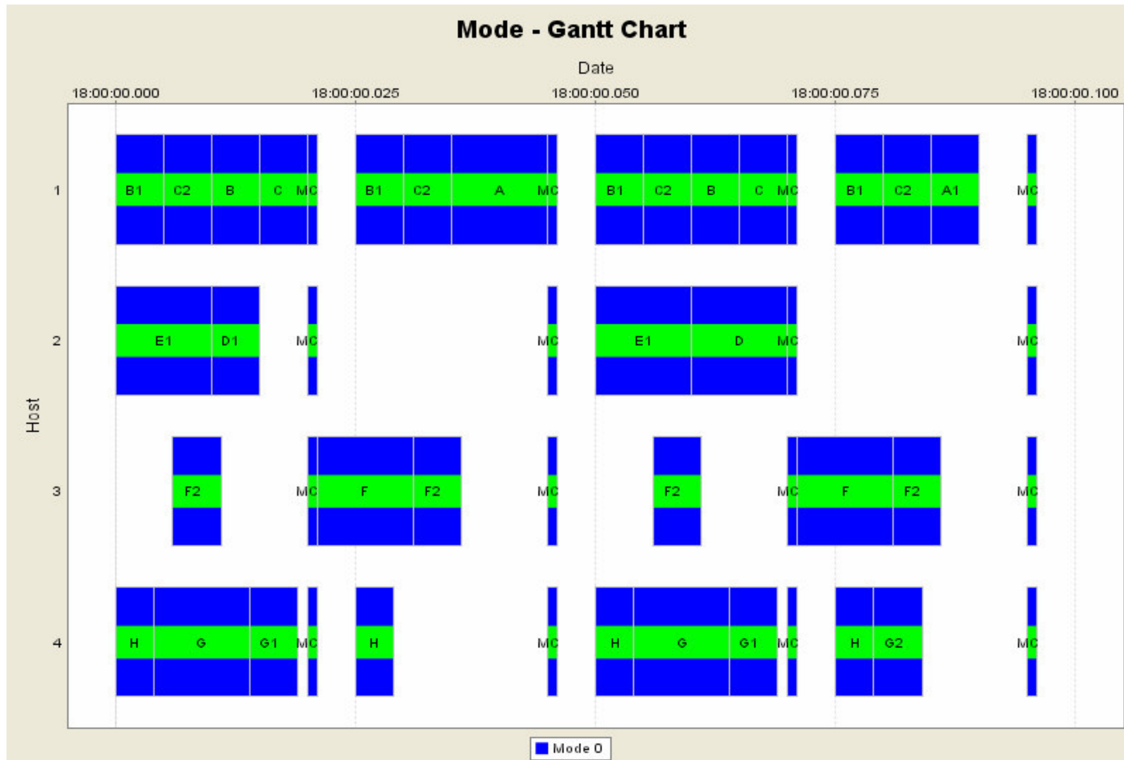

Figure 4.5 Gantt chart of schedule in Landing Mode

The above schedules are processed to generate the mode change schedule. On a mode change, the mode change schedule provides the entry point in the target mode. For the current data set, there are four instances of the MC task in each repetition window of the schedule for the Takeoff mode, which, implies that the aircraft control system can change its mode to the Cruise mode at any of these time instants. The entry point and the task schedule of the Cruise mode corresponding to each MC task in the Takeoff mode is shown in figures 4.7 to 4.10. The selected task is B (accelerometers) in the INU. The timing of task B is maintained across mode changes as seen in figures 4.7 to 4.10.

Figure 4.6 Gantt chart of schedules on all hosts in all modes

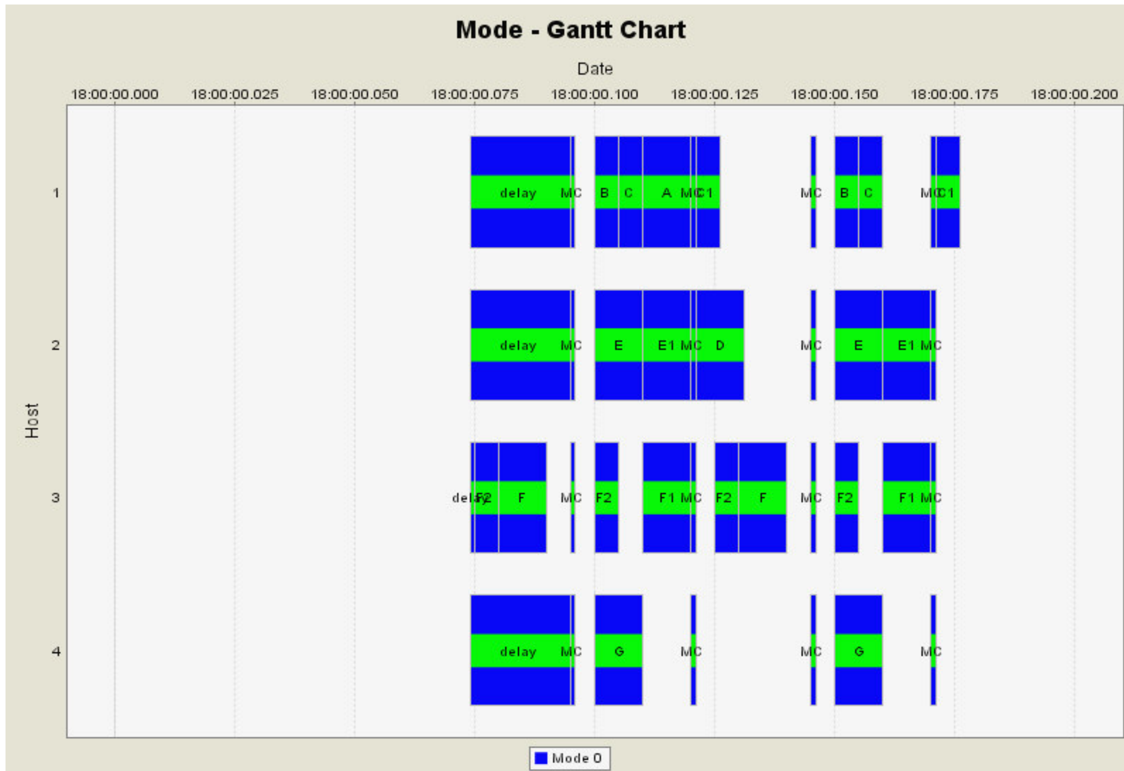

Figure 4.7 Schedule in Cruise mode corresponding to the 1st MC task (Takeoff Mode)

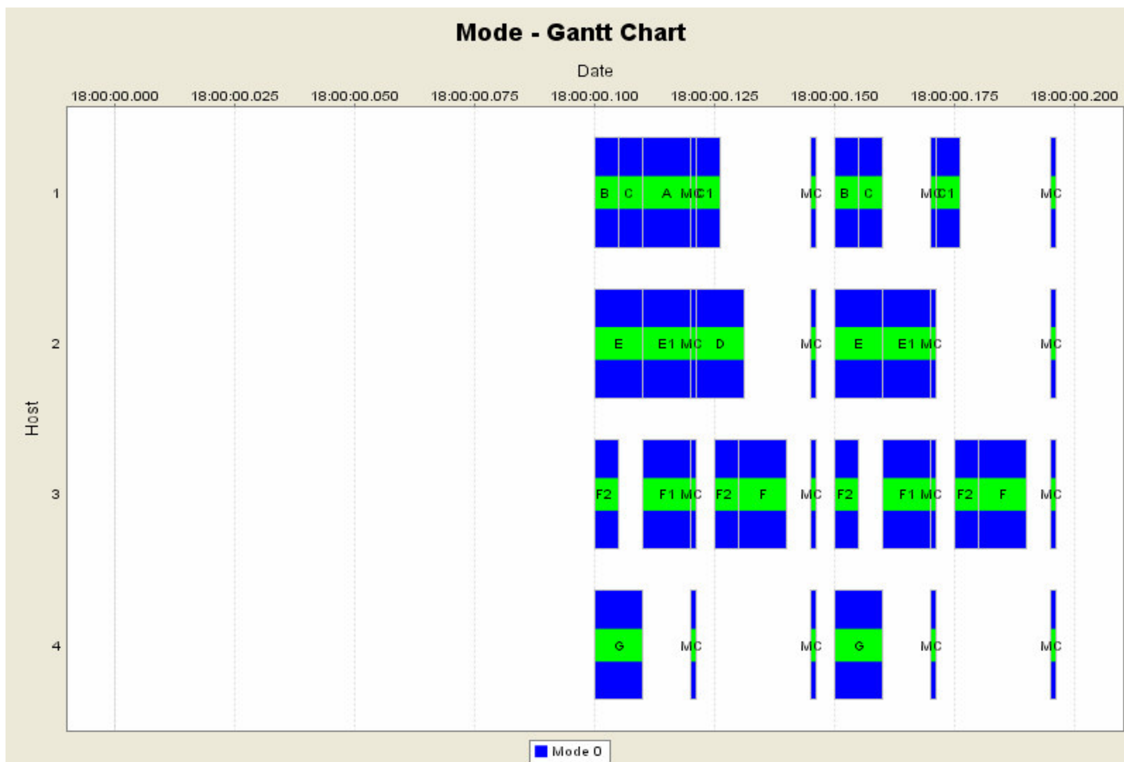

Figure 4.8 Schedule in Cruise mode corresponding to the 2nd MC task (Takeoff mode)

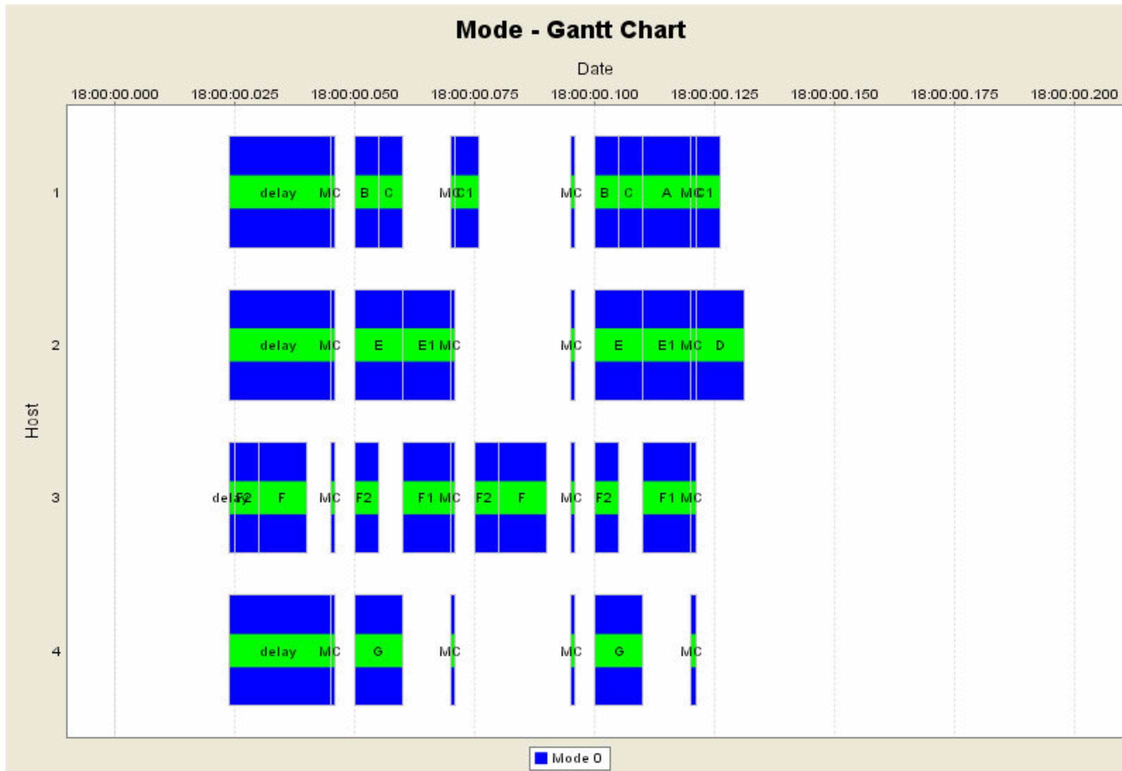

Figure 4.9 Schedule in Cruise mode corresponding to the 3rd MC task (Takeoff mode)

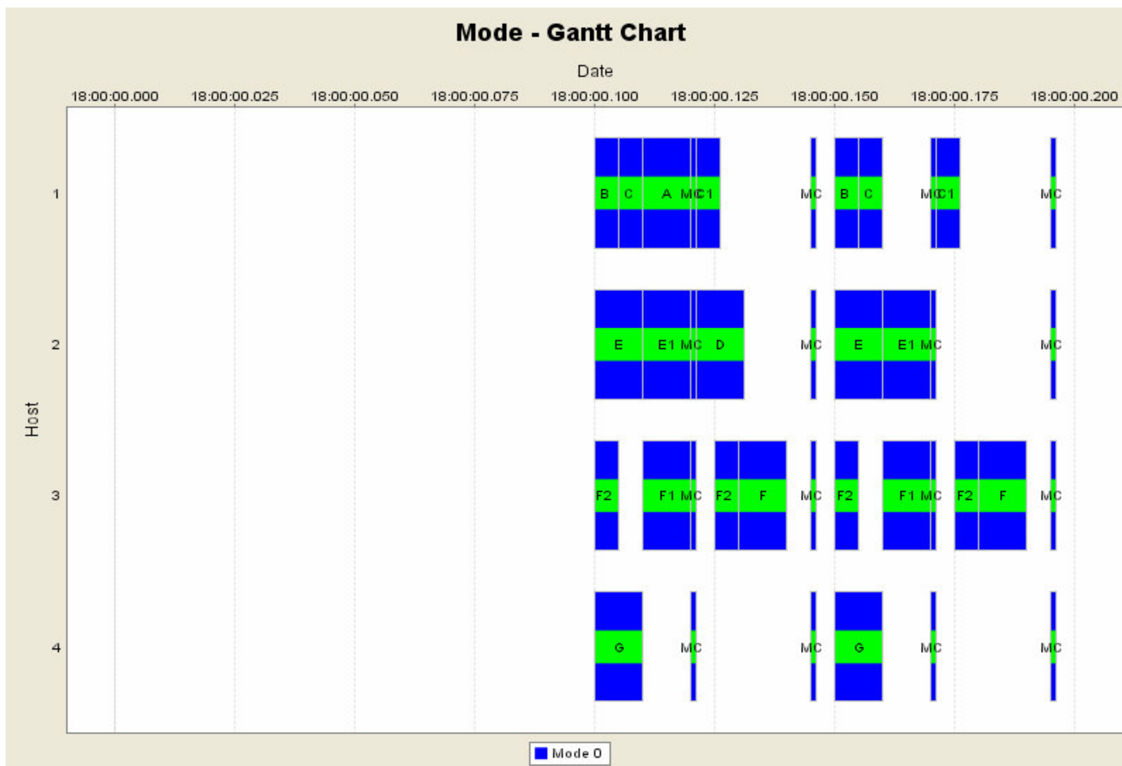

Figure 4.10Schedule in Cruise mode corresponding to the 4th MC task(Takeoff mode)

CHAPTER V

CONCLUSION AND FUTURE WORK

Control systems often experience changes in their operational modes. Scheduling the activities in such systems is very important for the reliable and predictive behavior of the physical system. A number of control systems today are designed based on the principles of time triggered architecture as TTA provides a reliable and fault tolerant architecture. With the increase in TTA implementations of control systems, scheduling in time-triggered systems undergoing mode changes has been recognized as an important and challenging area of research. This thesis presented an algorithm to generate a schedule which can handle mode changes in a predictable manner. The algorithm was implemented using constraint programming techniques. The implementation of the algorithm was verified by a case study using an aircraft control system. The aircraft system contains four operational units: the inertial navigation unit, pilot control system, global positioning system and an air data measurement system. The system operates in four different modes namely, takeoff, cruise, auto pilot and landing. The tasks in these different modes were successfully scheduled for execution. The individual schedules were processed successfully using the mode change algorithm to generate a mode change schedule.

Future Work

This thesis currently does not consider the role played by replicated nodes in a time triggered system while creating a schedule. Replicated nodes are used to detect and resolve faults [14]. Another important area for future research is to integrate the mode change schedule with the generation of schedule itself for individual modes. Work needs to be done to extend the current algorithm such that all the tasks executing in the previous mode maintain their timing across mode changes.

BIBLIOGRAPHY

[1] P. Baptiste, C. Pape, W. Nuijten, Constraint Based Scheduling: Applying Constraint Programming to Scheduling Problems, Kluwer Academic Publishers Boston 2001.

[2] K. Marriott and P. J. Stuckey, Programming with Constraints: an Introduction, MIT Press, Boston, 1998.

[3] R. Bartak *Constraint-Based Scheduling: An Introduction for Newcomers,* Technical Report TR 2002/2, Department of Theoretical Computer Science and Mathematical Logic, Charles University, 2002.

[4] K. Schild, J. Würtz, *Off-Line Scheduling of a Real-Time System,* In Proceedings of the ACM Symposium on Applied Computing, SAC98, ACM Press 1998.

[5] H. Kopetz, *The time-triggered architecture,* In Proceedings of International Symposium for ObjectOriented Real-Time Distributed Computing, pp. 22—29, 1998.

[6] J. Würtz, *Constraint-Based Scheduling in Oz,* In Operations Research Proceedings, pp 218-223, Springer-Verlag, 1996.

[7] C.Schulte, G Smolka and J Würtz, *Encapsulated Search and constraint programming in Oz*, In Second Workshop on Principles and Practice of Constraint Programming, Springer-Verlag, 1994.

[8] J. Real and A. Wellings, *Implementing Mode changes for shared resources in ADA.* In Proceedings of 11th Euromicro Conf. Real-Time Systems, pp 86-93, 1999.

[9] H. Thane and M. Larsson, *Scheduling using Constraint Programming*. Technical Report, 1997.

[10] C. Schulte, *Programming Constraint Services*, Lecture Notes in Computer Science, vol. 2302, Springer-Verlag Berlin Heidelberg 2002

[11] The Mozart Programming System, www.mozart-oz.org

[12] M. Fromherz *Constraint-based Scheduling.* American Control Conference (ACC'01), Invited Tutorial 2001.

[13] M. Ginsberg and W. Harvey, *Limited Discrepancy Search*. Proceedings of the Fourteenth International Joint Conference of Artificial Intelligence (IJCAI-95); Vol. 1, pp 607-613, 1995.

[14] H. Kopetz, *REAL-TIME SYSTEMS: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Boston, 1997.

[15] The ECLiPSe Constraint Logic Programming System, http://www.icparc.ic.ac.uk/eclipse/

[16] L. Sha, R. Rajkumar, and J.P. Lehoczky, "*Priority Inheritance Protocols: An approach to real-time synchronization*," IEEE Trans. Computers, vol. 39, no. 9, pp. 1175-1185, 1990.

[17] C. Pape, *Implementing of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-based Scheduling Systems.* In: Intelligent Systems Engineering, vol 3, 1994.

[18] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, *Giotto: A Time-Triggered language for Embedded Programming*. In Lecture Notes in Computer Science, Embedded Software. Heidelberg, Germany: Springer-Verlag, vol. 2211, pp. 166--184 2001.

[19] G. Fohler, *Realizing Changes of Operational Modes with Pre Run-time Scheduled Hard Real-Time Systems.* Responsive Computer Systems, Springer, 1993.

[20] M. Yokoo *Distributed Constraint Satisfaction: foundations of cooperation in multi-agent systems*, Springer 2001.

[21] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, *Embedded control systems development with Giotto,* In Proceedings of LCTES 2001.

[22] H. Kopetz, R. Nossal, R. Hexel, A. Krüger, D. Millinger, R. Pallierer, C. Temple, M. Krug, *Mode Handling in the Time-Triggered Architecture,* IFAC DCCS'97, Seoul, Korea.

[23] C. vanBuskirk, B. Dawant, G. Karsai, J. Sprinkle, G. Szokoli, K. Suwanmongkol, *Computer-Aided Aircraft Maintenance Scheduling,* ISIS Technical Report, 2002.

[24] E. Freuder, *In Pursuit of the Holy Grail*, In Constraints vol. 2, no. 1, pp. 57-61, 1997.