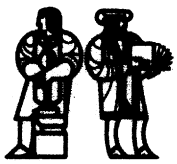
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-204

REAL-TIME CONTROL STRUCTURES FOR BLOCK DIAGRAM SCHEMATA

Thomas J. Teixeira

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

MIT/LCS/TR-204

REAL-TIME CONTROL STRUCTURES FOR BLOCK DIAGRAM SCHEMATA

by

Thomas Joseph Teixeira

August 1978

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

© Massachusetts Institute of Technology

Cambridge Massachusetts 02139

Acknowledgements

Steve Ward has been indispensable as an advisor in transforming my rather hazy ideas about real-time programming into a workable research topic. His enthusiasm kept this research progressing at many points when I was stuck or otherwise sidetracked.

Thanks are due to John Pershing, Al Mok and Jay Wahid for their work in providing a test bed for some of the ideas expressed in this thesis. Al Mok has also been especially helpful with his knowledge about scheduling theory and algorithms.

The entire Domain Specific Systems Research group has at some time contributed to the computer facilities that made the actual production of this document as well as the research possible, especially John Pershing and Terry Hayes.

Finally, I wish to thank Gillian Teixeira for her emotional and moral support throughout the course of this research.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

Table of Contents

1: Introduction 6.

1.1: Previous Work 6.
1.2: Statement of the Problem 10.
1.3: Thesis Overview 11.

2: Block Diagram Schemata 13.

2.1: Real-Time Performance and Specifications 15.
2.2: Functionality of Blocks 18.
2.3: Example 21.

3: Static Control Structures 25.

3.1: Existence of Cyclic Control Structures 27.
3.2: Generating Real-Time Control Structures 33.
3.3: A Branch-and-Bound Method for Generating Control Structures 38.
3.3.1: Determining the Relative Frequency of Constraint Paths 38.
3.3.2: Strategies for Combining Solutions 40.
3.3.3: Performance of the Algorithm 45.
3.3.4: Speeding up the Algorithm 48.
3.3.5: Practical Experience 50.
3.4: Heuristics for Generating Control Structures 52.

4: Static Priority Interrupt Control Structures 54.

4.1: Dynamic Control Structures 54.
4.2: Model for Static Interrupt Control Structures 56.
4.3: Assigning Priorities to Independent Tasks 60.
4.4: More Complex Models 62.
4.4.1: Scheduling Overhead 62.
4.4.2: Non-preemptive Control Structures 63.
4.4.3: Non-Distinct Priorities 63.
4.5: Applications to the Control Structure Problem 64.
4.5.1: Chains of Independent Tasks 64.
4.5.2: More Complex Task Relations 65.
4.5.3: Combining Static and Dynamic Control Structures 67.

5: Multiple Processor Control Structures 69.

5.1: Assigning Control Structures to Multiple Processors 72.
5.2: Dynamic Assignment of Processors 74.

6: Summary and Conclusions 75.

References 78.

List of Figures

2-1: A block diagram schema requiring a multi-processor control structure 17.
2-2: A Block Diagram Containing a Cycle 20.
2-3: Typical block diagram schema 21.
2-4: Latencies for static control structures 22.
2-5: Latencies for dynamic control structures with static schedulers 24.

3-1: Typical Laxity Table 31.
3-2: Counter-Example to Least Laxity Scheduling 37.
3-3: Block Diagram Where All Constraints Appear More Than Once 41.
3-4: Regions of a Critical Window 45.
3-5: Counter-Example to Slack as a Dominance Relation 46.

4-1: Counter-example to priority = 1 / latency 61.

5-1: A simple multi-processor control structure 70.

1: Introduction

There are many applications for computers where the real-time performance of the program is critical. These applications all involve asynchronous interaction with the external environment and it is this environment that imposes the real-time restrictions. For example, device drivers in operating systems must respond to interrupts before the information is lost. Another application is in direct digital control and process monitoring.

However, most high-level languages are not designed for producing time critical programs. The languages allow the user to define appropriate functional and data abstractions for his problem, but have no notion of real-time or asynchronous interaction with the real world. Instead, the user must design a control structure for his problem suitable for a single sequential process that will satisfy all the real-time constraints.

1.1: Previous Work

Many operating systems do have notions of real-time and external input and output, but they are supported at a fairly low level [19, 20]. The application program typically has to deal with priorities, setting real-time alarms, and responding to interrupts. These actions may be necessary to satisfy the constraints, but they do not bear a close relationship to the constraints. For example, it is seldom obvious what priority must be assigned to a task that must complete in ten milliseconds and uses one millisecond of CPU time.

Early work on applications oriented real-time operating systems was done by

Fiala [5]. Fiala proposed a model of real-time processes characterized by three parameters per process.

(1) P_i the maximum CPU time used by process i.

(2) D_i the maximum delay allowed from the time process i requests service to the completion of servicing that request.

(3) T_i the minimum period between requests for process i.

Fiala proposes three scheduling algorithms for this model. The first (and simplest) executes the process that must complete the soonest. i.e. the process with the earliest deadline. This algorithm is optimal in the sense that if any schedule satisfies the deadline requirements for all the processes, so does the earliest deadline schedule. However, this result is proved in the context of process switching requiring negligible overhead.

Fiala's second algorithm is a modification of the earliest deadline scheduler that minimizes the number of process switches while retaining the optimality condition of the earliest deadline algorithm. This is accomplished by having the scheduler check to see if the current process must be preempted when a process with an earlier deadline requests service. This is done by simulating the action of the scheduler on the current requests. Unfortunately, this algorithm would require extensive computation whenever a process requests service. Accordingly, Fiala's third algorithm pre-computes a lower bound on the expression required by the minimum switching algorithm. With the lower bound, the extra computation required by the third algorithm requires an extra comparison at process request time. The algorithm is also optimal in the same sense and requires less overhead than the simpler

earliest deadline algorithm.

However, Fiala makes no attempt to integrate his model and scheduler into a real-time language system. One such approach is control robotics developed by Dertouzos [3] and Geiger [6]. A control robotics program is organized as a set of daemons which continuously monitor some condition and execute the body (a corrective procedure) when the condition is true. The real-time specifications for a daemon are the delay from when a condition becomes true to when the program detects the condition (the recognition time) and the delay from detecting a condition and executing the body (the response time). Geiger's implementation of control robotics periodically samples the condition with a period slightly less than the recognition time (the slightly higher rate will allow for preemption by other daemon conditions). The daemon bodies are scheduled using an earliest deadline scheduler.

One weakness of control robotics is that no guarantee of satisfying the real-time constraints is made at compile time. This could be done if the user declared a minimum period between executions of a daemon body and the compiler determined the computation time of the daemon bodies. Since it is impossible to determine the computation time for an arbitrary procedure, the compiler may require declarations to determine the computation time.

A more substantial problem of Geiger's implementation is the assumption that the conditions for daemons are independent of the execution of other daemon bodies. Therefore, complex structures of daemons whose conditions depend on variables changed by other daemons could result in much unnecessary computation. All in all, control robotics does not provide any more of a model for real-time programming

than Fiala's work beyond suggesting some syntax for identifying tasks and specifying their deadlines.

Another system that deals with real-time specifications at the user level is TOMAL (Task Oriented Microprocessor Language) [12]. On the surface TOMAL is a combination of a modern block structured programming language and a typical mini-computer 'real-time' operating system. However, in addition to assigning static priorities to tasks, a response time may be specified for a task. This response time is similar to the recognition time for control robotics and specifies the maximum delay between a request for a task activation and the initiation of that task. Another feature of TOMAL is that interrupt routines only request task activation and do not respond to the interrupt in any substantive way. This reduces the amount of object code that does not run under the task scheduler and allows the TOMAL system to check the consistency of the real-time constraints for the entire system. However, TOMAL makes no attempt to verify real-time specifications on service times for tasks.

Data flow schemata deserve mention as a real-time system since one proposed applications is digital signal processing [2, 22]. It is designed to facilitate highly parallel computation and statements may be executed as soon as all their input variables have been computed. If several statements are executable an arbitrary statement is chosen. However, with the addition of real-time constraints to mediate this decision, data flow would be a powerful real-time system. The other major drawback of data-flow is that is not suited for implementation on conventional computer architectures.

1.2: Statement of the Problem

The goal of this research is to develop theory that is applicable to the implementation of a programming system designed to the restricted domain of time-critical applications. The main criterion of the suitability of the language to this domain should be that small changes in the real-time specifications should result in small, obvious changes in the source program. It is conceivable, and indeed desirable, that these changes could have a dramatic effect on the object program produced. This reorganization of the object program is precisely the process that should be automated.

Conventional languages already provide facilities for functional and data abstraction, and numerous researchers are already working in this area. Therefore, this research will focus on the global control structure for programs. This includes issues such as the number of processors to use in an implementation, deciding what interrupt structure (if any) is necessary, decomposing the program into tasks, and assigning parameters required by the appropriate task scheduler.

Since normal language semantic issues are being avoided, the description of a program can be made extremely simple. The intuitive model for a real-time program is that of continuous time analog block diagrams. The graph defines a precedence relation among operators identical to the data flow in the diagram. The program will be specified as a directed graph of actions to be performed and their functional dependence, with arcs of the graph representing data paths. The graph must be acyclic since cycles in a block diagram represent feedback systems. Automatically producing an object program that solves the feedback equation would require more detailed semantics for the programs as well as other disciplines outside the scope

of this research. However, in some special cases, cycles can be handles by rearranging the block diagram. A strict upper-bound must be placed on the computation time required for each action. The real-time constraints specify upper bounds of the propagation delays through the block diagram and of the bandwidths of the input and output signals.

1.3: Thesis Overview

Chapter 2 develops the *block diagram* model of computation. The block diagram model is a program schematic model similar to data flow. However, real-time and an external environment are explicit in the model. In addition, the block diagram model separates the data-flow of the schema from the control flow, which is embodied in the *control structure*. The control structure specifies the execution order of the blocks at object time. The research problem may be formalized as finding control structures for block diagram schemas which satisfy the given real-time specifications. The major use of the model is to define the semantics of the real-time specifications.

Chapter 3 investigates various static control structures (control structures that are independent of the data values at object time). Although static control structures may be used widely in specific applications (particularly in small, dedicated systems such as those implemented on microcomputers), they have been ignored by designers of real-time programming systems, mainly because their real-time performance in the general case has not been studied.

Chapter 4 investigates extended semantics where the external inputs do not change continuously. In this situation, a dynamic control structure may be used. A

dynamic control structure is a control structure that does depend on the data values at object time. The chapter investigates a subclass of dynamic control structures, namely static priority interrupt control structures. The prototypical example is an interrupt system where the system does nothing until an input changes, although it includes systems without physical interrupts where the inputs are sampled. The priorities are static as opposed to the earliest deadline scheduler where the priority of a task is a function of time.

Chapter 5 discusses some of the issues that arise when more than one processor is available for the implementation. The real-time performance of multiprocessor systems are analyzed and the real-time performance of a block diagram schema is bounded. Some techniques for distributing the processing among several processors are suggested, although specific algorithms are not studied.

2: Block Diagram Schemata

Most models of computation do not capture the notion of a "real-time" system which monitors continuously changing inputs from some external environment. Block diagram schemata model the external environment explicitly and recognize the existence of real-time specifications placed by the environment on the computing mechanism. They are based on the intuitive model of the conventional analog block diagram whose inputs and outputs are changing continuously. An *(m,n) block diagram schema* consists of an *(m,n) block diagram module*, a *control structure*, a *configuration* and an *environment* which manipulates the configuration asynchronously with the control structure. Within the model, it is assumed that values change continuously. Obviously, the computations cannot be performed continuously on a digital computer. The real-time specifications determine how often the control structure must compute new values, as well as how fast it must compute them.

An *(m,n) block diagram module* is a directed graph whose nodes are either *blocks* or *links*. The terms *predecessor* and *successor* will be used with the conventional definitions. Data is stored in the links while the blocks perform the actual computation. Accordingly, only one arc may point to each link. The graph must be proper in the sense that arcs may not point from links to links or from blocks to blocks. Upper-case letters will be used to denote blocks and lower case letters to denote links. The predecessor of a link is called the *specifier* of that link and the successors of a link are called the *watchers* of the link. The predecessors and successors of a block are called the *inputs* and *outputs* of the block respectively.

An (m,n) module has m links with no input arcs *(input links)* and n links with no output arcs *(output links)*. The input links receive their values from an external,

continuous time function called the *input signal*. The values at the output links define an external, continuous time function called the *output signal*.

The model assumes the existence of a global clock which defines the passage of real time. Hewitt argues against the use of global clocks since they cannot be implemented in distributed systems [9]. While Hewitt's objections against global clocks are valid, assigning times within Hewitt's framework of local orderings would be more complicated. This complexity is unnecessary since the events being timed are always ordered by one of Hewitt's local orderings.

A *configuration* is an assignment of *tokens* to the links of a schema. The token contains a value and a set of labels of the form (*link*, *birth*). These labels indicate when the token arrived at the input link *link*. Each link always contains some token, since signals are always defined in a continuous time block diagram.

The *computation* of a block diagram schema is described by a series of *snapshots*. A snapshot consists of a block diagram module and an associated configuration. The initial snapshot assigns null values to all tokens except for tokens on the input links of the schema which are assigned the current value of the input signal. The label set of all links is initialized to {(*link*, 0)}. The computation proceeds from one snapshot to the next through the *firing* of blocks. The *control structure* is the strategy for choosing which block to fire next. The fired block accesses the tokens on its input links, and replaces the tokens on its output links. The label set for the output token becomes the union of the old label set of the token and the label sets that were assigned to the tokens on all the input links of the block. The time in the label (*l*,*t*) for the link *l* at each input arc of the fired block is replaced by the label (*l*,*time*), where *time* is the current

contents of the global clock. This action occurs after any tokens have been replaced on the output links, but the time for the new label sets is immediately after the input tokens were accessed. In addition, if *l* is an input link, its value is set to the current value of the input signal. The block need not replace any output tokens. This differs from data flow since tokens are not removed from the input links after a block is fired. The data flow restriction is not appropriate since the value of a token is defined at all times.

The amount of computation time used by block *A* is denoted t_A. If the control structure fires block *A* on some processor at time *t*, that processor will complete and replace the output tokens on that block by the time $t+t_A$. The computation times used will be upper bounds either computed by whatever language processor is used to create the primitive blocks or declared by the user.

2.1: Real-Time Performance and Specifications

A block diagram schema is an approximation to a continuous time block diagram. There are many factors affecting the quality of the approximation. However, the factors influenced by the control structure are how long the schema takes to compute the values of output tokens from the input tokens, and how often it performs these computations. The real-time specifications will place bounds on these quantities. A control structure that satisfies all the real-time specifications is called a *feasible* control structure.

The *age* of a token with respect to a link *l* at time *t* is defined as $t-t_0$ if (l, t_0) is in the label set of the token, and undefined otherwise. The *latency* between

links a and b is denoted $l_{a,b}$ and is the upper bound of the age at any time of tokens at b with respect to link a. The user can specify an upper bound on the latency between two links. The first link will be an input link of the schema and the second link will be an output link.

Latency specifications can also be expressed in terms of continuous-time functions:

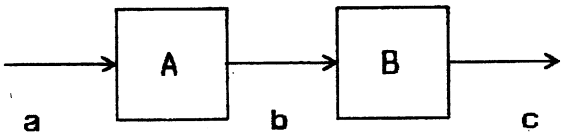
$$\hat{b}(t) = F(a(t-\Delta(t)), \cdots), \ \Delta(t) \leq l_{a,b} \qquad (2\text{-}1)$$

Here $\hat{b}(t)$ is the function whose value is the value of the token at link b at time t; $a(t)$ is the function whose value is the signal at link a at time t; $\Delta(t)$ corresponds to the age of the tokens at link b. Notice that $\Delta(t)$ is generally <u>not</u> constant, but is bounded. The user knows how close $\hat{b}(t)$ must be to $b(t) = F(a(t), \cdots)$. Using information about the magnitude of F and a and their derivatives, the user can use equation (2-1) to calculate the latency specifications necessary to achieve the desired accuracy of $\hat{b}(t)$.

The other measure of real-time performance is how often new values are computed. The bandwidth from link a to link b (notation $B_{a,b}$) is the maximum rate at which the control structure must compute new values at b from values at a. The bandwidth specification is not easily expressible in terms of continuous-time functions. It may be thought of as a requirement on how often the value of $\hat{b}(t)$ must change.

The bandwidth specification may seem superfluous since the latency specifications also implies how often the value of $\hat{b}(t)$ changes. However, it is possible for a multiple processor control structure to exhibit bandwidth performance

that exceeds the rate implied by the latency specification. An example is shown in figure 2-1.



$t_A = 10msec$
$t_B = 10msec$
$B_{a,c} = 75/sec$
$l_{a,c} = 40msec$

A block diagram schema requiring a multi-processor control structure
Figure 2-1

In this example, both *A* and *B* require ten milliseconds of computation time. A single processor control structure that executes *ABABAB* · · · can guarantee a latency from *a* to *c* of forty milliseconds and a bandwidth from *a* to *c* of fifty per second. However, if processor one executes *AAA* · · · and processor two executes *BBB* · · · , then the latency from *a* to *b* is still only forty milliseconds but the bandwidth increases to one hundred.

While the block diagram model is useful for defining performance for real-time programs, it does not yield many insights into the problem of synthesizing a feasible control structure. The graph itself resembles a partial order on a set of tasks, but the semantics of block diagram schemata are not as restrictive as this partial order. In most schematic models, a task must not be executed until all its predecessors have been executed since (presumably) it would not have data available at all its inputs. The block diagram model has no such restriction and as

a result is able to execute some parts of the schema more often than other parts.

On the other hand, there are certain execution orders that can be ruled out since they are obviously inefficient. For example, once a block has been fired, it need not be fired again until one of its predecessors has been fired again since all its inputs will be unchanged. Therefore, it outputs will not change. Similarly, if no successor of a block *A* is fired between firings of *A*, the previous execution of *A* was unnecessary since no block looked at the previous values of the tokens on the output links of *A*.

If these restrictions are combined, each firing of a block must be surrounded (in time) by at least one predecessor and at least one successor. Equivalently, the allowable execution sequences may be found by shuffling all the paths from an input link to an output link. These paths will be referred to as *constraint paths* or just *constraints*.

2.2: Functionality of Blocks

The semantics of block diagram schemata make some useful block functions awkward to implement. For example, a block that performs differentiation is essential for applications in real-time process monitoring and control. In classical direct digital control, the system is discretized by sampling at some specific period. Differentiators are replaced by unit delays and the feedback gains are adjusted appropriately. This is possible only because the inputs are sampled at a known frequency.

In block diagram schemata there is no guarantee of periodic execution. The bandwidth specifications set a lower bound on how often a block must be

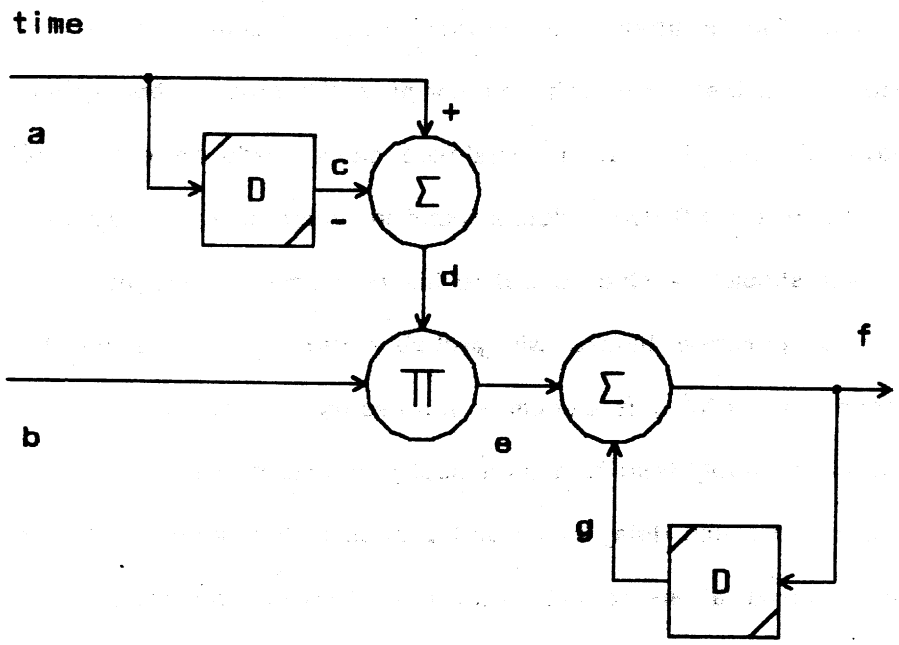
executed, and a different lower bound may be implied by the latency specifications. They do not place any upper bound on how often the block is executed. Therefore, it is impossible to tell *a priori* when and how often a block will be executed. This would seem to rule out any blocks that would require state variables, but this is not true. A white noise generator could be implemented using a pseudo-random number generator. This would use a state variable, but it would not run into any problems by not knowing how often it is executed. But most other functions that need to produce or transform a time dependent sequence of values will be impossible to implement.

The only general solution to the problem is to have a real-time clock as part of the system. Then a differentiation block could remember both its previous input and the time it was last executed and compute the obvious first order approximation. The major difficulty is that the real-time clock would have to provide much finer resolution than the 60 cycle clocks found in typical computer systems.

The user should be able to define his own time dependent functions since any selection of primitive blocks will probably turn out to be too limited for some application. Therefore, it becomes necessary to provide some primitive blocks which would probably lead to nonsensical programs if used carelessly. In particular, if the user had a unit delay block and access to the real-time clock he could define arbitrary approximations to differentiators, although undisciplined used of the unit delay block would result in useless programs.

Implementing integration would still be a problem since the block diagram for a first order integrator would contain a cycle (see figure 2-2). The problem with cycles is that it is unclear whether the cycle represents use of a state variable,

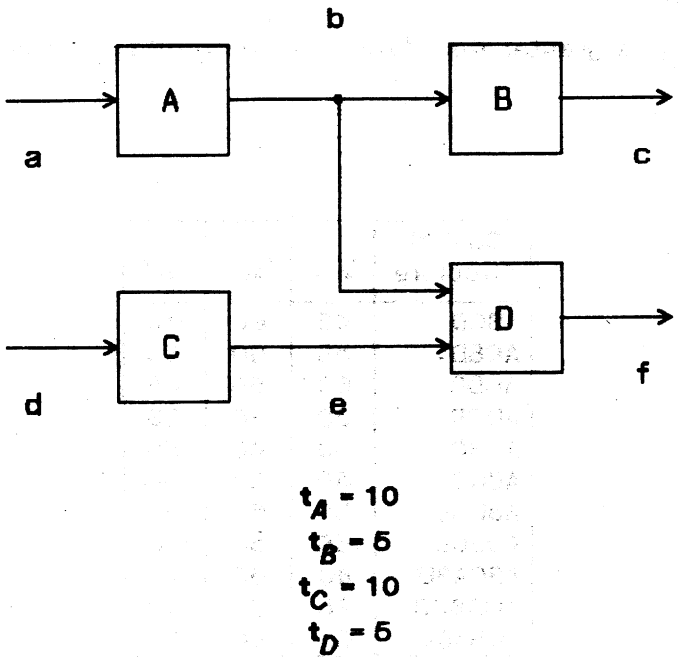
as in data flow, or implied solution of simultaneous equations, as in continuous time block diagrams. In the case of integrators it is clear that the cycle represents use of a state variable, since the cycle contains a unit delay block. In this case, the cycle can be broken at the input to the delay block. The delay block is treated as a watcher of link e, even though it gets its input from link *f*. This transformation alters the order in which the blocks are executed by changing the constraint paths. Unit delays were handled by a similar transformation in BLODI [11], a system for simulating discrete time block diagrams, and would be handled in the same way by a programmer [21].



A Block Diagram Containing a Cycle
Figure 2-2

2.3: Example

The interaction between the real-time specifications and the control structure can be illustrated by a series of examples. In these examples the block diagram module is left unchanged while the latency and bandwidth specifications are varied. These variations will necessitate changes in the control structure used to implement the block diagram schema. The block diagram module itself is shown in figure 2-3.



Typical block diagram schema
Figure 2-3

The simplest control structures to consider are cycles that repeatedly execute the blocks in some fixed order. There 3! (= 6) ways of executing four blocks once per cycle (ignoring starting transients). For a small example like this it is feasible to enumerate all such cycles and test them to see if they satisfy the latency

constraints[1]. All these control structures are independent of when new tokens actually arrive. The worse-case assumption is that a new token arrives immediately after the previous token is marked old. This assumption is used in calculating worst-case latencies, which are shown in figure 2-4. Notice that although ABCD is better than ACBD and ADBC is better than ADCB, there is no best control structure. In fact, we can choose latency specifications such that only one of the control structures will work. The first six control structures in figure 2-4 sample the inputs once per cycle, i.e. once every 30 time units. However, if any of the bandwidths $B_{a,c}$, $B_{a,f}$ or $B_{d,f}$ is greater than 1/30 then some other control structure must be used.

Control Structure	$l_{a,c}$	$l_{a,f}$	$l_{d,f}$
ABCD	45	60	45
ACBD	55	60	50
ACDB	60	55	45
ADCB	60	45	60
ADBC	50	45	55
ABDC	45	50	60
ABDCD	50	55	50
ADBCD	55	50	50
ABCABD	40	65	75
ACDBCD	75	70	40
ADBADC	65	40	70

Latencies for static control structures
Figure 2-4

A slightly more complicated class of control structures is cycles where some blocks may be executed more than once. For example, the control structure

1. However, such an algorithm is not practical since the computation time taken by such an algorithm would grow exponentially with the number of blocks.

ABCABD has worst-case latencies as shown in figure 2-4. This control structure will satisfy its bandwidth constraints if $B_{a,c}$ is less than one every twenty time units and $B_{a,f}$ and $B_{d,f}$ are less than one every forty-five time units.

The next class of control structures to consider are dynamic control structures with static priority scheduling. These control structures make use of the current environment to determine which blocks to fire next. The dynamic control structures assume that the values of tokens at input links do not change continuously. When the value of a token at an input link changes, a *request* is made for a set of *tasks*. The request is *serviced* by firing a fixed sequence of blocks as specified by the task. Since the processor is generally busy when a request occurs, the requests are remembered until the processor is idle, when one of the requested tasks is selected to be executed. Each task is assigned an integer *priority*. The task with the highest priority is serviced next. The scheduler is static since the priority for a task is always the same relative to other tasks. The earliest deadline scheduler is an example of a dynamic priority scheduler, since the priority of a task depends on its current deadline. If the task being serviced can be temporarily suspended, the control structure is preemptive.

A dynamic control structure need not be interrupt driven. For example, the control structure could sample the inputs between executing blocks. However, preemptive control structures cannot be implemented without interruputs.

In the example of figure 2-3, there are many ways to construct tasks to be requested by changing inputs. One such task system is to fire *ABD* (or *ADB*) when the value at *a* changes, and *CD* when the value at *d* changes. The worst case occurs when the values at *a* and *d* change simultaneously. The latencies for this

case are shown in figure 2-5. These latencies can be sustained only if the bandwidths at a and d are both less than once every 35 time units (otherwise the control structure would fall behind). In a sustained worst case, new tokens arrive once every 35 time units. A trace of block firings would seem to indicate that the static control structure ABDCD is being executed, which has latencies 15 to 20 units larger than those for the dynamic control structure. However, in the dynamic case it is known exactly when the input signal change. In particular, the processor will be idle if more than 35 time units elapse between a change in input signals, so the processor will be able to respond to a change immediately. In a static control structure, the change would not be responded to until the control structure gets around to it.

Task String		$l_{a,c}$	$l_{a,f}$	$l_{d,f}$
Priority 1	2			
ABD	CD	30	35	15
ADB	CD	35	30	15
CD	ABD	15	20	35
CD	ADB	20	15	35

Latencies for dynamic control structures with static schedulers
Figure 2-5

3: Static Control Structures

The main function of the control structure in a schema is to specify when to fire each block. If the control structure is independent of the configuration (i.e. unaffected by changes made by the environment) it is a *static* control structure. An example of a static control structure is a loop which fires all of the blocks in the schema cyclically. Control structures which make use of configuration (e.g. via interrupts) are called *dynamic* control structures.

The latency specification from *a* to *b* will be satisfied only if all the blocks along all paths from *a* to *b* are fired at least once during each time interval of duration $l_{a,b}$ time units. Otherwise there would be time intervals longer than $l_{a,b}$ when the *a*-label at *b* will not change and therefore the age with respect to *a* of the token at *b* will be greater than $l_{a,b}$. Similarly, the bandwidth specification from *a* to *b* will be satisfied if and only if the interval between firing the blocks along the constraint paths is less than $1 / B_{a,b}$.

For single processor control structures it is possible to construct a *trace* of the blocks that are fired by the control structure. The trace is a string over an alphabet Σ whose elements correspond to the blocks of the schema. Each element *A* of Σ is assigned a *weight* (notation $|A|$) equal to t_A. The weight of a string is defined to be the sum of the weight of its elements. A string S_1 *contains* S_2 if all the elements of S_2 appear in S_1 in the order they appear in S_2. For example, the string *ABCDE* contains the string *BD*, even though *BD* is not a substring of *ABCDE*. Regular expressions will be used to denote sets of strings. In particular, if *S* is a string, S^* denotes the set of strings *S*, *SS*, *SSS*, $\cdots$ as well as the empty string.

It is necessary to model intervals in continuous time of arbitrary origin and duration, since the latency specifications require <u>all</u> intervals of specific duration to

contain the corresponding constraint path. Therefore the weight of the initial and final elements of a string may be counted at less than their nominal weights. For example, if $|a_1 \cdots a_k| = w$ (weighting a_1 and a_k at $|a_1|$ and $|a_k|$), then $[a_1 \cdots a_k]$ is a string of weight less than w since both a_1 and a_k are weighted at less than $|a_1|$ and $|a_k|$. However, if the initial cr final elements do not have full weights, the may not be included as part of any contained string. Weighting these elements at less than their full values corresponds to shrinking an interval of size w in continuous time: if the interval starts after a_1 starts executing, then the interval does not contain a_1 reading its inputs. A string will be preceded by a '[' or followed by a ']' if the first or last element in the string is weighted at less than its nominal value.

A single processor static control structure is completely specified by its trace, which is determined at compile time (hence the name *static* control structure). The real-time specifications on the control structure can be rephrased as constraints on its trace. In particular, the latency specification from a to b is satisfied if and only if all the constraint paths from a to b are contained in every substring in the trace of weight $l_{a,b}$. The bandwidth specification is satisfied if and only if the weight of all substrings between occurrences of the constraint paths are less than $1 / B_{a,b}$.

At this point it is possible to deal exclusively with the trace of the control structure and the constraint paths. Constraint path i will be denoted C_i with latency specification l_i and bandwidth specification B_i. If C_i is a path from a to b, $l_i = l_{a,b}$ and $B_i = B_{a,b}$. It will also be necessary to deal with the tails of the

constraint paths. If $C_i = c_{i,1}c_{i,2}\cdots c_{i,n}$, where $c_{i,j}\epsilon\Sigma$ then the jth tail of C_i is $C_{i,j} = c_{i,j}c_{i,j+1}\cdots c_{i,n}$.

Since the control structure must satisfy the real-time specifications for all time, the trace corresponding to the control structure will be a infinitely long string. Since the control structure can be implemented only if the trace can be generated using a finite program, it would be very awkward if the only feasible control structures were acyclic. Fortunately, it can be proved that if any feasible control structure exists, then there exists a feasible control structure that fires the blocks in some cyclic order.

3.1: Existence of Cyclic Control Structures

The theorem proved in this section can be stated as:

Suppose there exists a string $\omega = a_1a_2a_3\cdots\epsilon\Sigma^*$ such that ω satisfies the real-time constraints. Then there also exists a finite string β such that the string β^* also satisfies the real-time specifications.

This theorem will be proved using several lemmas.

Definition: A *critical window* of a control structure ω for the constraint C_i is a substring $\psi_i = a_k\cdots a_m$ of ω that contains two occurrences of C_i, but $[\psi_i]$ contains no occurrences of C_i.

The *most critical window* for C_i is the critical window with the greatest weight.

Lemma 3-1: The string ω satisfies the latency specifications for C_i if and only if $|\psi_i| \le l_i$ for the most critical window ψ_i in ω.

Proof:

only if: Assume ω satisfies the real-time specifications. Then any substring of ω of weight l_i contains C_i. In particular, the substring $[a_k \cdots a_m a_{m+1}]$ of weight l_i must contain C_i. Since $[\psi_i]$ does not contain C_i, the substring $[\psi_i$ of weight $l_i - \epsilon$, where ϵ is arbitrarily small contains one occurrence of C_i. Therefore, $|\psi_i| \leq l_i + \epsilon$, $\epsilon > 0$.

If: Assume the most critical window ψ_i has weight greater than l_i. Let γ be any substring of $[\psi_i]$ where $|\gamma| = l_i$. γ exists since:

$$|[\psi_i]| = |\psi_i| - \epsilon > l_i - \epsilon$$

Since ψ_i is a critical window, then $[\psi_i]$ contains no occurrences of C_i. But γ is a substring of $[\psi_i]$ and also does not contain C_i. Hence, γ is a substring of ω of weight l_i that does not contain the constraint path. Therefore, ω does not satisfy the latency specifications. ■

Corollary: Since ψ_i contains two occurrences of C_i, the period between successive occurrences of C_i must be less than $l_i - |C_i|$.

This lemma shows there is a time limit between the starts of successive occurrences of C_i. The bandwidth specifications directly limit this interval. Therefore, it will be assumed that the latency specifications are more severe than the bandwidth specifications. If not, the latency specifications can be adjusted so that:

$$l_i \leq \frac{1}{B_i} + |C_i|$$

The time remaining until the start of the next appearance of a constraint path is called the *laxity* of that constraint. Given a control structure, we can construct a table of laxities for each position in the corresponding string ω with the property that the table entries are non-negative if and only if ω satisfies the latency specifications. The only difficulty is in accurately determining the start of an occurrence of a constraint string. This will be handled by keeping laxities for the

tails of the constraint strings. The true laxity for a string will be reflected in the laxities of its tails if the start of the constraint path is falsely identified.

An element of the table $d[i,j,k]$ is the laxity for the path $C_{i,j}$ just before a_k is fired. The table should be thought of as rectangular with columns labeled by elements of ω. The entries in the first column are:

$$d[i,j,0] = l_i - |C_{i,j}| \tag{3-1}$$

since the constraint path C_i must occur by $l_i - |C_{i,j}|$. The remaining columns can be filled in by simple recursion rules.

If the next element in ω is not the same as the first element in a constraint path, the laxity for that path decreases by the weight of that element:

$$a_k \neq c_{i,j} \Rightarrow d[i,j,k+1] = d[i,j,k] - |a_k| \tag{3-2}$$

There are two possibilities if the next element in the solution is the same as the first element in a constraint path. If this <u>is</u> the start of an occurrence of a constraint path, the laxity for the tail of that path should be no more than the current laxity for the constraint path. It is possible that the tail will already have a more severe laxity since different constraint paths can have identical tails. In addition, the laxity for the whole constraint path will become the original limit the instant after the first element appears. Therefore, the laxity becomes the original laxity minus the weight of the first element.

However, if a_k is <u>not</u> the start of an occurrence of $C_{i,j}$, the laxity should decrease by $|a_k|$. Fortunately, this problem will be handled automatically by assuming that an occurrence of $C_{i,j}$ starts whenever $a_k = c_{i,j}$. If it is not part of an occurrence of $C_{i,j}$, $c_{i,j}$ will appear again before all of $C_{i,j}$ appears. When this

happens, the laxity for $C_{l,j+1}$ will have decreased by the amount the laxity for $C_{l,j}$ should have decreased if the start of the path had not been incorrectly identified. When $c_{i,j}$ appears again, the laxity for $C_{i,j+1}$ will be less than the laxity for $C_{i,j}$. Therefore:

$$a_k = c_{i,j} \Rightarrow \begin{cases} d[i,j+1,k+1] = min(d[i,j,k], d[i,j+1,k] - |a_k|) \\ d[i,j,k+1] = l_j - |C_{i,j}| - |a_k| \end{cases} \quad (3\text{-}3)$$

Equations (3-2) and (3-3) can be transformed to produce rules for computing the k+1st column of the laxity table from the kth column:

$$d[i,j,k+1] = \begin{cases} l_j - |C_{i,j}| - |a_k| & \text{if } a_k = c_{i,j} \\ min(d[i,j-1,k],\ d[i,j,k] - |a_k|) & \text{if } a_k = c_{i,j-1} \\ d[i,j,k] - |a_k| & \text{if } a_k \neq c_{i,j}, c_{i,j-1} \end{cases} \quad (3\text{-}4)$$

As an example, figure 3-1 shows the laxity table for the control structure *ABCD* and the block diagram module from figure 2-3.

In this table, the laxities at time 60 are identical to the laxities at time 30. The next column in the table would be identical to the column at time 40. The rest of the table becomes periodic, and all the entries are non-negative. The periodicity allows us to prove that $(ABCD)^*$ will satisfy the latency specifications for all time. This is formalized in the following lemmas:

Lemma 3-2: If:

$$\forall_{i,j}\ d[i,j,m] \geq d'[i,j,k] \text{ and } a_k = a'_m$$

then:

	A (0)	B (10)	C (15)	D (25)	A (30)	B (40)	C (45)	D (55)	A (60)	... (70)
AB	30	20	15	5	0	20	15	5	0	...
B	40	30	35	25	20	0	35	25	20	...
CD	30	20	15	20	15	5	0	20	15	...
D	40	30	25	15	35	25	20	0	35	...
AD	45	35	30	20	15	35	30	20	15	...
D	55	45	40	30	50	15	10	0	50	...

$$t_A = 10 \; t_B = 5 \; t_C = 10 \; t_D = 5$$

$$l_{AB} = 45 \; l_{CD} = 45 \; l_{AD} = 60$$

Typical Laxity Table
Figure 3-1

$$\forall_{i,j} \; d[i,j,m+1] \geq d'[i,j,k+1]$$

Proof: From case analysis of (3-4) and elementary algebra. ■

Lemma 3-3: Let:

$$\alpha = a_1 \cdots a_{k-1}$$

$$\beta = a_k \cdots a_{m-1}$$

$$\gamma = a_m \cdots$$

If $\omega = \alpha\beta\gamma$ satisfies the latency specifications and:

$$\forall_{i,j} \; d[i,j,k] = d[i,j,m]$$

then:

$$\omega' = \alpha\beta\beta\gamma$$
$$= a'_1 a'_2 \cdots$$

also satisfies the latency specifications.

Proof: Construct the laxity table d' for ω':

$$\forall_{i,j} \; d'[i,j,0] = l_j - |C_{i,j}| = d[i,j,0]$$

Since $a_1 = a'_1$, (3-4) leads to:

$$\forall_{i,j}\, d'[i,j,1] = d[i,j,1]$$

Similarly:

$$\forall_{i,j,l\le m}\, d'[i,j,l] = d[i,j,l] \ge 0 \qquad (3\text{-}5)$$

Therefore

$$\forall_{i,j}\, d'[i,j,m] = d'[i,j,k]$$

From lemma 3-2:

$$\forall_{i,j}\, d'[i,j,m+1] \ge d'[i,j,k+1]$$
$$= d[i,j,k+1] \ge 0$$

Similar reasoning will show:

$$\forall_{i,j}\, d'[i,j,2m-k-1] \ge d'[i,j,m-1]$$
$$= d[i,j,m-1] \ge 0$$

Now $a'_{2m-k} = a_m$, so lemma 3-2 still applies:

$$\forall_{i,j}\, d'[i,j,2m-k] \ge d[i,j,m] \ge 0$$

Inductively:

$$\forall_{i,j,l\ge m}\, d'[i,j,l+m-k] \ge d[i,j,l] \ge 0 \qquad (3\text{-}6)$$

Combining (3-5) and (3-6):

$$\forall_{i,j,l}\, d'[i,j,l] \ge 0$$

Therefore, from lemma 3-1, ω' satisfies the latency specifications. ■

Corollary: Let $\alpha = a_1 \cdots a_{k-1}$, $\beta = a_k \cdots a_{m-1}$, and $\gamma = a_m \cdots$. If $\omega = \alpha\beta\gamma$ satisfies all the latency specifications and $d[i,j,k] = d[i,j,m]$ for some $k<m$, then $\alpha\beta^*$ also satisfies the latency specifications. The proof is by induction. ■

The main theorem can now be proved by showing that any laxity table will have duplicate columns and applying lemma 3-3:

Theorem 3-4: If any string ω satisfies the latency specifications then there exists a string of the form β^* which also satisfies the latency specifications.

Proof: Construct the laxity table for ω. There are a finite number of possibilities for each table entry since each entry is $l_i - |C_i|$ minus a sum of a finite number of $|a_k|$'s. The number of different $|a_k|$'s is limited by the number of blocks in the block diagram schema. The number of terms in the sum must be finite since each $|a_k|$ is greater than zero and the laxity entry is also greater than or equal to zero. Therefore, the possibilities for each column are limited and eventually some column in the table will be repeated and k and m satisfying the conditions of lemma 3-3 exist.

Applying the corollary to lemma 3-3 says a solution of the form $\alpha\beta^*$ exists. However, $d[i,j,1] = l_i - |C_{i,j}| \geq d[i,j,k]$, for all k (the rules for filling in the table never increase the laxities except to set $d[i,j,k]$ to $l_j - |C_{i,j}|$. Applying lemma 3-2 shows that β^* is also a solution. ■

The major implication of this theorem is that only cyclic strings need to be considered for static control structures. These strings can be enumerated, so the problem of finding a static control structure is in principal solvable. Since the proof also places an upper bound on the length of the cycle (equal to the total number of possible laxities at any position), so an algorithm that generated all possible strings would be effective in the sense that it would always halt in a finite amount of time. However, it would require computation time that grows exponentially with the complexity of the schema, so the problem would be computationally intractable if this were the only algorithm.

3.2: Generating Real-Time Control Structures

The problem of generating a feasible control structure is a scheduling problem. The problem is deterministic since the parameters of the problem are strictly bounded as opposed to being unbounded random variables. A wide varieties of special cases of the general scheduling problem have been studied, and some results are surveyed by Gonzalez [7], though relatively little work has been done

on scheduling in the presence of deadlines.

Gonzalez and Soh developed a simple algorithm that minimizes the number of processors used to schedule independent tasks. The tasks are statically assigned to processors and always run to completion. The deadlines for each task correspond to the period of the requests for that task and must be a power of two. Their algorithm is not optimal if the periods are not a power of two and no optimal algorithm is known, although several heuristic algorithms have been investigated.

Liu and Layland considered the problem of scheduling independent tasks on a single processor [14]. Each task requests service periodically with a deadline for service coinciding with the time for the next request. They present a method of assigning static priorities to the tasks that will meet the deadlines if any static assignment of priorities will. In addition, they prove the schedule which executes the task whose deadline is earliest is optimal in the sense it will meet the deadlines if any schedule will. They then prove necessary and sufficient conditions for a set of tasks to be scheduled by the earliest deadline (ED) algorithm to meet all its deadlines, and conclude that ED algorithm allows 100% utilization of the processor as opposed to figures as low as 70% for static priority algorithms.

Geiger extended the proof of the optimality of ED scheduling to include the case were the requests are not periodic [6]. Fiala presented the same basic proof and also derived necessary and sufficient conditions for the ED scheduler with a mix of periodic and aperiodic tasks [5].

Mok investigated scheduling independent tasks on multiple identical processors [16]. Mok shows that no optimal algorithm exists for this problem unless the

deadlines, computation times and at least some future request times are known. An algorithm related to the ED algorithm is presented which is shown to be optimal if all requests are simultaneous. This algorithm executes those tasks with the least laxity, where the *laxity* of a task is the deadline for the task minus its remaining computation time. Unfortunately, both the least laxity and ED schedulers are shown to be non-optimal even for tasks with periodic requests. However, the least laxity scheduler is optimal for periodic deadlines where tasks may be executed at any time (i.e. if the deadlines are coincident with the next request, the least laxity scheduler is optimal if it is allowed to execute tasks before they have been requested).

The problem of scheduling tasks related by a partial order on multiple identical processors has been studied by Manacher [15]. Deadlines are specified for any or all tasks in the system. Manacher's algorithm derives deadlines for all tasks in the system by using the observation that a task must complete executing in time to allow its successors to executed before their deadlines. The scheduler then executes those tasks with the earliest deadlines that have had all their predecessors executed. This algorithm is not optimal, and does not consider either periodic requests or multiple start-times. However, it is a reasonable heuristic, especially as the number of processors increase.

Unfortunately, none of these results generalize to the static control structure problem, even for a single processor, although control structures could be constructed which would meet the conditions of the particular special case and satisfy the real-time constraints. For example, if the block diagram consisted of unconnected (independent) blocks, the earliest deadline scheduler could be used

with task *i* being block *i* and the request period for each task being the minimum of $l_i / 2$ and $1 / B_i$. The period between requests would have to be less than $l_i / 2$ since (in the absence of other information) it is possible for the task to be executed immediately after one request and immediately before the following deadline. Lemma 3-1 says this time interval must not be greater than l_i.

On the other hand, these heuristics areliable to be overly restrictive, particularly since they tend to deal with independent tasks. It would be possible to derive independent tasks from a block diagram schema by treating the constraint paths as independent, but at the cost of introducing new blocks and much unnecessary computation. One promising approach for deriving a static control structure is to simulate some more general control structure until a cycle in the trace of that control structure is found. An obvious choice of a more general control structure is a least laxity scheduler (using laxities as defined for block diagram schema) which follows the partial order for the tasks (blocks) based on the constraint paths. More precisely, the scheduler would build a laxity table, with starred entries indicating constraints strings which cannot be fired because of the partial order. The scheduler chooses the first block of the unstarred constraint string with the smallest laxity to head the next column. If two constraints have the same laxity, either can be fired next. Figure 3-2 shows such a laxity table for the block diagram schema from figure 2-3 using the same latency specifications as figure 3-1.

At time 40, none of the latency specifications have been violated. However, since there are now two constraints with laxity 0, at least one entry in the next column will be negative. By firing *C* at time 10, an additional request for *C* is

	A (0)	C (10)	D (20)	B (25)	A (30)	? (40)	... ...
AB	30	*20	*10	*5	0	*20	...
B	40	30	20	15	35	0	...
CD	30	20	*20	15	10	0	...
D	40	30	20	35	30	20	...
AD	45	*35	*25	20	15	*35	...
D	55	45	35	50	45	15	...

$t_A = 10$ $t_B = 5$ $t_C = 10$ $t_D = 5$

$l_{AB} = 45$ $l_{CD} = 45$ $l_{AD} = 60$

Counter-Example to Least Laxity Scheduling
Figure 3-2

created with deadline 50. In the control robotics environment, the existence of this request makes scheduling impossible. However, if *B* is fired and *C* is delayed until time 15, the additional request also gets delayed to a point where it is possible to schedule all the requests. The least laxity algorithm simply does not deal with interactions between requests and deadlines.

It is interesting to note that the least laxity scheduler fails for this even if the constraint path *AD* is ignored. The remaining constraint paths *AB* and *CD* are independent, yet they cannot be scheduled using the ED algorithm using the worst-case period of $l_i / 2$. If periods are kept at $l_i - |C_i|$, the tasks still cannot be scheduled by the ED scheduler if the individual blocks are scheduled separately. The failure in this case can be viewed as an inability of the ED scheduler to derive the proper phase relation between the tasks.

The schedule shown in figure 3-3 is not the only least laxity schedule. For example, at time 25 *CD* has the same laxity as *B* and therefore *C* could be fired instead of *B*. However, the reader can verify that all the least laxity schedules for

this example fail to satisfy the latency specifications.

3.3: A Branch-and-Bound Method for Generating Control Structures

Rather than generating acyclic control structures and looking for a cycle, the algorithm described in this section works by generating a cyclic control structure that satisfies the real-time specifications for one of the constraint paths. The solutions for other constraints paths are combined to form a control structure that satisfies all the real-time specifications. The basic semantics of firing blocks rules out control structures that are not shuffles of the constraint paths since these control structures perform redundant computations. Therefore, this algorithm should not miss any solutions. There are two major problems that the algorithm has to deal with: (1) How many times must each constraint path appear in one cycle of the total control structure. (2) How should the constraints paths be combined into one cycle.

3.3.1: Determining the Relative Frequency of Constraint Paths

The first step in the algorithm is to determine how many times each constraint appears in one cycle of the total solution. Upper and lower bounds can be derived from the length of the cycle and the basic latency specification. Consider the lower bound on the number of appearances of constraint i: let k_i be the number of appearances of C_i in one cycle of the solution ω^*. Let $w_i = |C_i|$ and $c = |\omega|$. Since the latency specification for C_i requires C_i to appear at least once every

$l_i - w_i$ time units:

$$k_i \geq \frac{c}{l_i - w_i} \tag{3-7}$$

This leaves c (the length of the cycle) to be determined. However, if C_i appears k_i times:

$$c \geq k_i w_i \tag{3-8}$$

More precisely, the algorithm starts with the assumption that each block and constraint appears once and that $c = \sum_A t_A$. This approximation is used to derive k_i for all constraints in the schema. If any k_i increases, this is used to update the minimum number of times each block in the constraint must appear, which in turn may cause c to increase. This process continues until all k_i are consistent with c. In practice, this only takes a few iterations.

Theorem 3-4 places an upper bound on the number of blocks in a cycle, but this bound is not directly applicable to the branch and bound algorithm since the branch-and-bound algorithm does not try all cycles of a given length. An upper bound on the number of appearances of any constraint can be easily derived if the number of appearances of the other constraints is held constant.

First, an upper bound on the length of a cycle can be derived by applying equation 3-7 to all constraints except constraint i. Then the minimum weight of a cycle containing k_j appearances of C_j can be computed for all $i \neq j$. Letting c_{max} be the maximum allowed cycle weight and c be the minimum cycle weight (not including constraint i), the minimum weight of a cycle containing k_i appearances of

C_i is:

$$c + k_i w_i \tag{3-9}$$

Therefore, the upper bound on k_i can be derived by restricting the resultant cycle weight to be less than c_{max}:

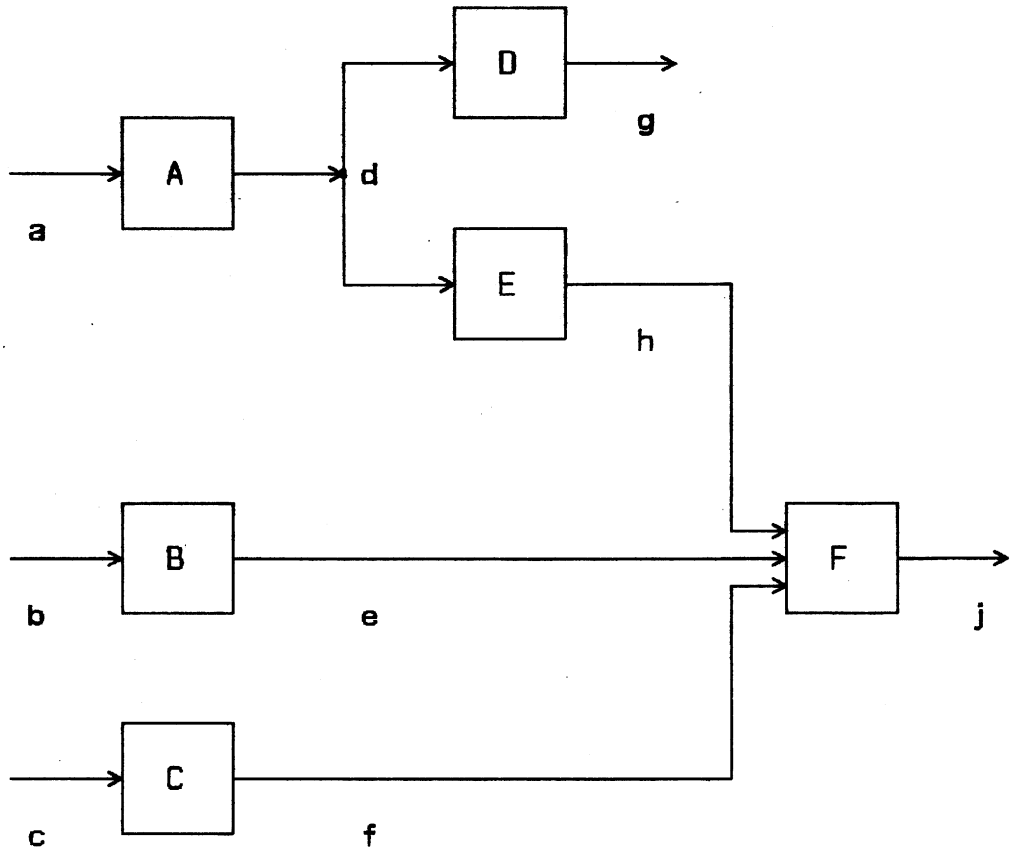
$$k_i \leq \frac{c_{max} - c}{w_i} \tag{3-10}$$

This ignores the possibility of blocks in C_i already appearing in the cycle as part of other constraints. However, including more appearances of constraint *i* will eventually cause the minimum cycle length to exceed c_{max}.

This still does not bound the number of appearances for all constraints, since constraint *i* can appear more often if constraint *j* appears more often, etc. Placing an arbitrary bound on one constraint will also bound the number of appearances of all other constraints. For example, requiring at least one constraint to appear only once places a fairly tight bound on all constraint. However, it is not true that a solution of this type always exists. An example is shown in figure 3-3.

3.3.2: Strategies for Combining Solutions

Once the number of appearances per cycles of each constraint path is known, the constraint paths can be permuted to form a control structure which satisfies all the real-time specifications. Many of the techniques for improving the efficiency of 'branch-and-bound' optimization algorithms can be applied to this problem even though it is not an optimization problem. An optimization problem seeks a



$$t_A = t_B = t_C = t_D = t_E = t_F = 1$$

$$\begin{aligned} l_{a,g} &\leq 11 \\ l_{a,j} &\leq 15 \\ l_{b,j} &\leq 7 \\ l_{c,j} &\leq 10 \end{aligned}$$

Control Structure: $(ABFDECBFADEBFCF)^*$

Block Diagram Where All Constraints Appear More Than Once
Figure 3-3

permutation of n objects that maximizes an evaluation function f of the permutation.

A 'branch-and-bound' algorithm for this problem generates permutations for a

subset of the objects and extends these permutations to larger subsets. The permutations to the subsets are called *partial solutions*, and are arranged in a tree. Nodes in the tree correspond to partial solutions and the descendents of a node are the extensions of that partial solution. Branch-and-bound algorithms are often more efficient than direct enumeration since it is often unnecessary to examine the entire search tree. The key to pruning the search tree is the dominance relation on nodes of the tree. The evaluation function f can be extended to arbitrary nodes of the search tree by defining the value of a non-terminal node to be the maximum value of its descendants. Then node A dominates node B if and only if $f(A) > f(B)$. The branch-and-bound algorithm may prune any subtree whose root node is dominated by some node of the tree that has already been explored.

In general, the dominance relation for a particular optimization problem cannot be computed without examining the entire tree. However, it is often easy to compute some weaker relation. These weaker relations are usually referred to as *dominance relations* in the literature, so we will use the term *strong dominance relation* to refer to the dominance relation that relates A to B if and only if $f(A) > f(B)$.

Branch-and-bound algorithm vary in the order the tree is searched and how the dominance relations used to prune the search tree. Kohler and Steiglitz classified branch-and-bound algorithms and initiated the theoretical study of dominance relations [13]. They demonstrated the surprising result that pruning based on a stronger dominance relation does not always improve the efficiency of the algorithm. However, Ibaraki showed that stronger dominance relations do lead to more efficient

algorithms for several common classes of branch-and-bound algorithms [10].

Branch-and-bound algorithm as defined by Kohler and Steiglitz also make use of a function g that places a upper bound on the value of f at each node. If L is the maximum $f(A)$ for leaf nodes A encountered, pruning sub-trees with $g(A) \leq L$ can only improve the efficiency of the algorithm. However, the upper bound function can also be viewed as a particular dominance relation.

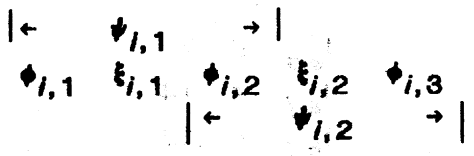
The control structure problem as stated is not an optimization problem. However, it is still possible to define a dominance relation between nodes of the search tree: node A strongly dominates node B unless B leads to a valid control structure and A does not. Assuming the nodes at each level are generated in a random (lexicographic) order, the best pruning for the algorithm to use is to retain the node at each level which dominates the other nodes. If this dominance relation can be easily computed, the algorithm can generate a valid control structure without backtracking.

As a first step towards computing a dominance relation, define the *slack* for each constraint to be the difference between the latency requirement and the latency actually achieved by the control structure. The constraint with the least slack is the *most critical constraint (MCC)*. The slack in the MCC could also be used as a value function to be maximized. If no control structure satisfies the real-time constraints, the control structure maximizing the slack in the MCC is probably a good 'close' solution. Also, the slacks may be used to evaluate any heuristic algorithms for deriving control structures.

The latency achieved by a static control structure for a constraint C_i is the weight of the most critical window for C_i. Adding a block to the cycle of the

control structure cannot increase any slacks since the weight of some critical window will be increased. The only exception would be if the new block completes an additional occurrence of some constraint path, thereby creating new critical windows. This cannot happen if the blocks being added are elements of some other constraint path, since no constraint path is contained in another constraint path. Therefore, the MCC slack can be used as an upper bound function in a branch-and-bound algorithm to maximize the MCC slack. Upper bound functions are also often used to guide the search in branch-and-bound algorithms. For example, the algorithm could always expand the node with the greatest upper bound.

If the slacks in each constraint are reduced by the same amount when a new block is added to the cycle, then the partial solution with the greatest MCC slack would be a dominant solution. Unfortunately, this is not generally the case. Consider dividing a cycle ω of the control structure ω^* into regions $\phi_{i,j}$ and $\xi_{i,j}$, as shown in figure 3-4. The $\phi_{i,j}$ regions contain one occurrence of C_i, but $[\phi]$ contains no occurrence of C_i. The critical windows of C_i are $\phi_{i,j}\xi_{i,j}\phi_{i,j+1}$. Therefore, adding blocks to a $\xi_{i,j}$ region increases the weight of $\psi_{i,j}$, and adding blocks to a $\phi_{i,j}$ region increase the weight of $\psi_{i,j-1}$ and $\psi_{i,j}$. Even if $|\psi_{i,j}|$ increases, the slack for C_i will not decrease unless $|\psi_{i,j}| = \max_k |\psi_{i,j}|$. The slacks can not be used to compute a dominance relation since the interdependence of constraint paths may force new blocks to be added within the most critical window of some constraint, while another solution with a smaller MCC slack might have a critical window of the right size in the right place.



Regions of a Critical Window
Figure 3-4

Keeping vectors of slacks for each constraint path does not correct the problem. Consider the example shown of figure 3-3 with the latency specification as shown in figure 3-5. It can be easily verified that $(ADEFCADBC)^*$ is a feasible control structure for this schema. It is also the only feasible control structure[1]. *AD* and *CF* must appear at least twice in one cycle of the solution. Figure 3-5 shows slacks for this constraints for two partial control structures. The merging of $(ADAD)^*$ and $(CFCF)^*$ that leads to the solution is $(ADFCADFC)^*$. However, the slacks for *CF* in $(ADCFADCF)^*$ are larger and the slacks for *AD* are the same, so $(ADCFADCF)^*$ would dominate $(ADFCADFC)^*$ even though it doesn't lead to a solution.

3.3.3: Performance of the Algorithm

Assume each constraint path contains an average of k blocks. The slack of a constraint path in a trial cyclic solution can be determined in at most k scans of the cycle. If there are n constraint paths there will be $o(nk)$ scans of each trial solution generated by the algorithm. The trial cycles will be $o(nk)$ blocks long (this

1. This was verified by checking all cyclic control structures that might be generated by a branch-and-bound algorithm assuming that the least critical constraint only appears once per cycle.

$$l_{a,g} \leq 7$$
$$l_{a,j} \leq 14$$
$$l_{b,j} \leq 12$$
$$l_{c,j} \leq 10$$

Control Structure	Slack Constraint			
	AD	CF	AEF	BF
ADFCADFC	1	2	-	-
ADCFADCF	1	4	-	-

Counter-Example to Slack as a Dominance Relation
Figure 3-5

ignores the possibility of a constraint appearing several times in one cycle). The overall time complexity of the algorithm will be $o(n^2k^2)$ times the number of trial cycles generated per problem.

Assume the trial cycle contains m_1 blocks and the next constraint path contains m_2 blocks. There are $(m_1+m_2-1)!$ cycles containing all the blocks, but we are only interested in one of the $m_1!$ permutations of the blocks in the old cycle, and $(m_2-1)!$ permutations of the blocks in the new constraint (i.e. we must consider m_1 different phase relations of the two cycles). Therefore, the number of different trail cycles generated at this step is:

$$\frac{(m_1+m_2-1)!}{m_1!(m_2-1)!} = m_1 \binom{m_1+m_2-1}{m_1} \qquad (3\text{-}11)$$

Of course, if some blocks of the new constraint are already contained in the old cycle, or if the next constraint appears more than once, not all of the generated cycles will be distinct. However, it is rather difficult to avoid generating these

cycles. There will be relatively little extra cost to the algorithm as long as it does not investigate cycles that are identical to cycles that have already lead to failures. Therefore, the number of trial cycles generated by the merging algorithm when it finds a solution without backtracking is approximately:

$$\sum_{l=2}^{n} k\binom{kl-1}{k} \tag{3-12}$$

Equation (3-12) is $o(kn^{k+1})$ since the binomial term in the sum is $o(n^k)$ and there are n terms.

If the merging algorithm fails to find a solution, then it must have backtracked through each trial solution and the total number of cycles generated is:

$$(1+k\binom{2k-1}{k}(1+\cdots(1+k\binom{nk-1}{k})\cdots) \tag{3-13}$$

which can be approximated:

$$\prod_{l=2}^{n} k\binom{kl-1}{k} \tag{3-14}$$

Equation (3-14) is $o((kn^k)^n)$ or $o(k^n n^{kn})$, and is exponential in the number of blocks in the schema. This is a very loose upper bound and would only be achieved if all generated solutions were plausible except when the last constraint was being merged in. However, this bound is achievable if the first $n-1$ constraint paths had relatively large latency specifications while the last constraint path had relatively small latency specifications. This situation can be easily avoided by starting with the path with the smallest latency constraints relative to the weight of the path.

3.3.4: Speeding up the Algorithm

There are many ways the average performance of the algorithm could be improved. For example, if we had a tighter lower bound on the slack in the MCC, we could prune more subtrees. We can get a tighter bound by determining what new blocks must be added to the control structure. Adding a new block always increases the size of some critical window for a constraint by at least the weight of the block. Therefore, if the sum of the slacks for a constraint is less than the total weight of blocks that must be added to the control structure, at least one of the critical windows for that path will exceed the latency specification for that path. This tighter bound has no effect on the performance if no backtracking is necessary. However, if no solution is found, using the tighter bound is roughly equivalent to reducing n, since fewer constraints need to be combined before the control structure is recognized as infeasible.

Notice that the performance of the algorithm would not be of polynomial complexity even if there were a dominance relation that totally ordered the possibilities at each level. The problem is that the number of partial solutions that must be generated by a naive algorithm can grow exponentially with the complexity of the schema. Therefore, finding a good dominance relation is not as important as finding a search function that generates nodes that are most likely to lead to a solution first.

Since the weight of the critical windows increase when new blocks are added, we might try merging in new constraint paths so that no new blocks are added before trying more general mergings. This will improve the performance if the solution is an extension of this type of merging, even if the algorithm must

backtrack since fewer nodes are generated on that level. If the algorithm must backtrack through all the control structures of this type, the performance of the algorithm is somewhat worse. The effect of this heuristic may be approximated by reducing k, since the length of the strings merged into the current control structure will be reduced.

The other way of improving the performance of the algorithm is to reduce the complexity of the problem. This can be done by replacing sub-graphs of the block diagram module with new blocks. Whenever the new block is fired, the blocks comprising the subgraph replaced by the new block are fired in some fixed order. This replacement can dramatically reduce k, and would improve both the best- and worst-case performance. However, combining blocks in this way can result in a schema which has no feasible control structures even though the original schema does.

Since the process of generating a control structure can be so time consuming, it would be extremely useful to quickly identify real-time specifications that are impossible to satisfy. One way of doing this is to compute the percentage of CPU time required by each block. If the sum of this percentage over all blocks in the schema is greater than 100%, the latency specifications are obviously unsatisfiable.

The percentage of the CPU required by each block is easily computed: each constraint C_i must be executed at least once every $l_i - |C_i| + \epsilon$ time units. Therefore, each block $c_{i,j}$ in C_i must be executed at least once every $l_i - |C_i| + \epsilon$ time units and its corresponding CPU percentage is:

$$\frac{|c_{i,j}|}{l_j - |C_j| + \epsilon} \tag{3-15}$$

If an block appears in several constraints, its CPU percentage is the maximum of the percentage implied by each constraint the block appears in. Using the maximum rather than the sum corresponds to assuming that each time the block is fired it will help satisfy all the constraints it appears in. Although this is not necessarily the case, it is a lower bound on the CPU usage.

Another quick test for unsatisfiable latency specifications is that the slack in each latency specification must be larger than the computation time for all blocks not contained in that constraint path. Otherwise, the ϵ portion of some critical window for that constraint will be too large (refer to figure 3-4).

3.3.5: Practical Experience

A branch-and-bound algorithm similar to the one described above has been implemented as part of a system for implementing continuous-time block diagrams on conventional micro-processors. The implementation runs on a PDP-11/70 under the UNIX timesharing system. The block diagram is described using an interactive graphics editor developed by John Pershing [18]. The branch-and-bound algorithm is only responsible for choosing the order to execute the blocks. The object code for the block diagram is produced by a separate program.

The program uses all of the heuristics mentioned above except it does not combine sub-graphs into new blocks. The program is able to find control structures to satisfy most latency specifications for small block diagrams using less than a minute of CPU time. So far, only one set of latency constraints has been found

where a valid control structure exists but no control structure was found by the program (see figure 3-3). Some latency specifications require more time to find a valid control structure.

In the absence of a fast optimal algorithm, it is preferable to have a fast algorithm which yields 'good' control structures quickly. Heuristic algorithms are generally evaluated one of two ways: one approach chooses a fixed algorithm and derives an upper (or lower) bound on how far the algorithm's solution is from the optimal solution. For example, Graham's algorithm for scheduling independent tasks on multiple processors executes tasks which require more processing time first. The resulting schedule is no more than 4/3 times as long as the optimal schedule [8].

The other approach develops a family of algorithms each requiring polynomial time. As the degree of the polynomial increases, the solutions found by the programs are closer to optimal. The family of algorithms is monotonic in the sense that the an algorithm taking more time never produces a poorer solution than one taking less time. If the degree of the polynomial were increased to infinity the algorithm would be optimal. However, it would also no longer be polynomially time bounded. An example is a series of scheduling algorithms employing limited lookahead [1].

The second approach does not seem applicable to the control structure problem. Limiting the breadth of back-tracking yields a family of exponential time algorithms with the exponent increasing with the amount of back-tracking. A family of polynomial algorithms would result if at most k blocks were merged at a time with no backtracking. However, these algorithms are very unsatisfactory if any

constraint must appear more than once. If the number of blocks in the constraint path is less than k, than all blocks for the second (and subsequent) appearance of the constraint will be merged coincident with the existing occurrences of those blocks. If k is increased so this does not happen, the performance of the algorithm is only slightly better than the complete algorithm with no backtracking.

3.4: Heuristics for Generating Control Structures

Steve Ward has experimented with some quick, simple heuristics for generating static control structures. Basically, the heuristic constructs control structures of the form $(\alpha\beta\alpha\gamma\alpha\delta\cdots)^*$ where α is the most critical constraint path and β, γ, δ, et cetera are taken from the other constraint paths. More specifically, blocks from the next most critical constraint are added to β with the restriction that $|\alpha\beta\alpha|$ is less than l_α. If more blocks remain in the constraint they are added to γ so that $|\alpha\gamma\alpha|$ is less than l_α. Once all constraints have been merged in this way, the latency specifications are checked. If they are all satisfied then the generated string is a feasible control structure.

The heuristic will also call itself using the current solution as α so the generated solution may also be of the form:

$$((\alpha\beta\alpha\gamma\cdots)\delta(\alpha\beta\alpha\gamma\cdots)\varepsilon\cdots)^*$$

Since these heuristics construct a control structure rather than search for one, they run very quickly. However, they also do not find solutions to a fairly large number of latency specifications, even for simple block diagrams. Still these heuristics are more attractive as a basis for an approximate algorithm, not only

because of their speed but also these heuristics could be extended to handle particular styles of block diagrams as the process of constructing control structures becomes better understood.

4: Static Priority Interrupt Control Structures

In some applications, the tokens at the input links do not change continuously. If the control structure can detect when an input changes, the real-time performance can be improved. Intuitively, this is possible since if no inputs to a block have changed, that block does not need to be executed. On the average, this type of control structure ought to do less computation and therefore ought to have better real-time performance. On the other hand, better average performance does not guarantee better worst-case performance and specific questions of performance must be answered with respect to a particular model.

Although the prototypical example of a dynamic control structure is interrupt driven, it is important to realize that hardware interrupts are not necessary. For example, a control structure could sample the inputs until one or more inputs change. After all the computation initiated as a result of these changes had completed, the control structure would continue to sample the inputs. In general, such a scheme would risk missing changes in the inputs. However, the control structure can use the real-time specifications to guarantee this will not happen.

4.1: Dynamic Control Structures

Many of the strategies for scheduling independent tasks to satisfy real-time constraints mentioned in the previous chapter use dynamic control structures. For example, Liu and Layland use static priority interrupts and consider the case (in our terms) where the latency is equal to the period between requests [14]. They consider the earliest deadline scheduler only in this context although the earliest deadline schedule is optimal for any sequence of requests and deadlines, as mentioned earlier.

Given an optimal scheduler, is there any reason to consider a suboptimal

scheduler? The answer will be yes if a good suboptimal scheduler exists which uses less resources than the optimal scheduler. The earliest deadline scheduler needs to find the highest priority task to execute whenever a task completes (alternately, it needs to insert requests into the proper position in a task queue). A static priority interrupt control structure also needs to find the highest priority task to execute. However, this is done in hardware by many existing computers, including current microcomputers. Also, the earliest deadline scheduler requires a real-time clock to compute the deadlines for each task from the request time and the latency specification. Therefore, static interrupt control structures are sufficiently simpler than a earliest deadline control structure to deserve further consideration.

4.2: Model for Static Interrupt Control Structures

A static interrupt control structure associates a *task* with each block in the diagram. The tasks are related by a precedence relation consistent with the block diagram. Each task has a *priority* and may be *idle*, *active*, or *requested*. The priority may be thought of as an integer with numerically greater priorities being better.

When an input changes, all tasks whose blocks are watchers of that input become requested. The control structures chooses the task with the highest priority among the requested tasks. This task is active until the block complete executing when all its successor tasks become requested and the task itself becomes idle. If the control structure allows active tasks to be suspended while another task is executed the control structure is call *preemptive*. Otherwise it is

non-preemptive. Unless otherwise noted control structures are assumed to be preemptive.

The latency performance of any static interrupt control structure can be determined for each task by adding the computation time for that task to the maximum computation time used by higher priority tasks while the task is on the ready queue. The difficulty in this analysis is in determining how much computation might be used by other tasks.

The simplest case to consider is when all the tasks are *independent* (each task consists of exactly one block). Each task i requires t_i units of computation; and has priority p_i, latency l_i, and bandwidth B_i. Without loss of generality, the tasks can be numbered so that:

$$p_1 \geq p_2 \geq \cdots$$

The overhead of associated with interrupts, selecting a task for execution, etc. will be ignored for the time being. We shall also assume that all priorities are distinct.

The latency for task i when its inputs change discretely is simply the maximum elapsed time between a change in an input and the termination of the task. This must be less than l_i if the latency specification for task i is satisfied. The interpretation of the bandwidth specification is also simplified. Instead of specifying a minimum rate for sampling inputs, the bandwidth specifies the maximum rate at which an input changes.

The latency specification for task i will be satisfied if and only if the block for task i can be completely executed during any time interval of duration l_i. During

this interval, tasks with priority better than p_i will also be run, and the amount of CPU time used by higher priority tasks must be less than $I_i - t_i$.

Notice that this model is equivalent to the model used by Fiala. Fiala's P_i corresponds to t_i, D_i corresponds to I_i, and T_i corresponds to $1/B_i$. Therefore, for a single processor we have the obvious restrictions:

$$t_i \le I_i \le \frac{1}{B_i} \tag{4-1}$$

and:

$$\sum_{i=1}^{n} B_i t_i \le 1 \tag{4-2}$$

The summands in (4-2) are the fraction of CPU time used by task *i*. Obviously the total fraction of the CPU used by all the tasks must be less than one. Equation (4-1) can be derived from (4-2).

Lemma 4-1: The amount of CPU time used by *n* independent tasks using a static priority scheduler in a window of duration Δt does not depend on the relative priority of the tasks.

Proof: The processor is always busy if some task is requesting service. Changing the priorities of the tasks will never cause the processor to remain idle when some task requests service, nor will it affect when the tasks request service.

Since the control structure only executes a task if some input to the task changes, task *i* cannot be executed more often than once every $1/B_i$ time units. Clearly, a task uses the maximum CPU time if any interval if it requests service at this maximum rate.

Assume task *i* requests service at times 0, $1/B_i$, $2/B_i$, $\cdots$, and let $C_i(t)$ be the maximum amount of CPU time used by task i in the interval $(0, t)$. The highest

priority task (task 1) always starts executing immediately after it requests service and executes for t_1 time units, so it will be executed $\lfloor B_1 t \rfloor$ complete times in the interval. Let $r = t - \lfloor B_1 t \rfloor$ be the amount of time at the end of the window after the last request for task 1. Task 1 will be executing during the interval $(t-r, t)$ since task 1 has the highest priority. However, if $r \geq t_1$, only t_1 units of computation will be used so:

$$C_1(t) = \lfloor B_1 t \rfloor t_1 + min\left(t_1, t - \frac{\lfloor B_1 t \rfloor}{B_1}\right) \tag{4-3}$$

The maximum amount of CPU time used by task 1 in the interval $(\Delta t, t+\Delta t)$ is:

$$C_1(t+\Delta t) - C_1(\Delta t) \tag{4-4}$$

We will show that this is maximized when $\Delta t = 0$ by showing:

$$C_1(t+\Delta t) - C_1(\Delta t) \leq C_1(t)$$

or

$$C_1(t+\Delta t) - C_1(t) \leq C_1(\Delta t) \tag{4-5}$$

Since the requests for task 1 occur with a regular period, $C_1(t)$ is also periodic. In fact:

$$C_1(t+1/B_1) = C_1(t) + t_1 \tag{4-6}$$

Therefore, we need only consider Δt between 0 and $1/B_1$, in which case:

$$C_1(\Delta t) = min(t_1, \Delta t) \tag{4-7}$$

This is the maximum amount of CPU time used by any interval of duration Δt since the CPU time used cannot be greater than the duration of the interval nor

can it be greater than t_1 if the interval contains less than one period. Therefore, the inequality in (4-5) holds since the left hand side is the amount of CPU time used in an interval of duration Δt starting at t.

The worst case for a set of tasks will occur when all tasks request service at time 0 and continue requesting service at their respective maximum rates. This is true since the highest priority task will use its maximum amount of CPU time under these conditions, and by lemma 4-1, any task can be made the highest priority task without affecting the amount of CPU time used by the set of tasks.

Define $C_j(t)$ by:

$$C_j(t) = \left\lfloor B_j t \right\rfloor t_j + min\left(t_j,\ t - \frac{\left\lfloor B_j t \right\rfloor}{B_j}\right)$$

The amount of CPU time used by tasks j and k is not necessarily $C_j(t)$ summed over j and k. The difficulty is that if requests for tasks j and k occur sufficiently near the end of the window and of each other then only the higher priority task will actually be executed. Therefore, it is necessary to determine a precise schedule for the interval from 0 to t. However, if we are only interested in how much CPU time is used in this interval, lemma 4-1 assures us that we may assign arbitrary priorities to tasks j and k.

However, a sufficient condition for satisfying the latency specification for task i is:

$$l_i \geq t_i + \sum_{j=1}^{i-1} C_j(l_i) \qquad (4\text{-}8)$$

This equation can be made more intuitive if the time required by task j is approximated by:

$$I_j B_j t_j \tag{4-9}$$

Then equation (4-2) becomes:

$$I_j \geq t_i + I_j \sum_{j=1}^{i-1} B_j t_j \tag{4-10}$$

This can be rewritten as:

$$I_j \geq \frac{t_i}{1 - \sum_{j=1}^{i-1} B_j t_j} \tag{4-11}$$

The denominator in equation 4-11 represents the fraction of CPU time available to task *i*. The effect of higher priority tasks is equivalent to reducing the CPU speed.

4.3: Assigning Priorities to Independent Tasks

One of the weaknesses of traditional real-time operating systems based on static priority scheduling is that the system does not verify that the priorities assigned by the user are consistent with his real-time specifications. Even if the system checked these specifications, the user still must assign priorities, which do not have a simple relation to the real-time specifications. The obvious strategy of assigning the highest priority to the task that requires the fastest response time does not work. Consider the example in figure 4-1. Either task 1 or task 2 can run at the best priority since $I_j \geq t_j$. If $p_j = 1/I_j$, then $p_1 > p_2$ and the the latency for task 2 is:

$$t_2 + \lfloor I_2 B_1 \rfloor t_1 + min\left(t_1, I_2 - \frac{\lfloor I_2 B_1 \rfloor}{B_1}\right)$$

$$= 12 + \left\lfloor \frac{16}{4} \right\rfloor 2 + min\left(2, 16 - \left\lfloor \frac{16}{4} \right\rfloor 4\right)$$

$$= 12 + 8 + min(2,0)$$

$$= 20 \not\leq I_2 = 16$$

However, the latency for task 1 if $p_2 > p_1$ is:

$$t_1 + \lfloor I_1 B_2 \rfloor t_2 + min\left(t_2, I_1 - \frac{\lfloor I_1 B_2 \rfloor}{B_2}\right)$$

$$= 2 + \left\lfloor \frac{15}{24} \right\rfloor + min\left(12, 15 - \left\lfloor \frac{15}{24} \right\rfloor 24\right)$$

$$= 2 + 0 + min(12,15)$$

$$= 14 \leq I_1 = 15$$

$t_1 = 2$	$B_1 = \frac{1}{4}$	$I_1 = 15$
$t_2 = 12$	$B_1 = \frac{1}{24}$	$I_1 = 16$

Counter-example to priority = 1 / latency
Figure 4-1

The algorithm successively finds a task that can satisfy its latency specifications while assigned the lowest priority. If there are several such tasks, choose one arbitrarily. This task is assigned the lowest priority and removed from the set of tasks. The next task selected will be assigned a priority higher than all previously assigned priorities but lower than all tasks still unassigned. This continues until no task remains or no task can be found that can execute at a

priority lower than all other tasks. In this case, no assignment of static priorities will satisfy all the latency specifications using only one processor. This algorithm will never make a bad choice. Consider the situation when one or more tasks remain yet no task can be assigned the lowest priority. Any task that could possibly run at a lower priority has already been assigned a lower priority.

4.4: More Complex Models

The model for static interrupt control structures made several simplifying assumptions, such as ignoring scheduling overhead, assuming preemptive scheduling and distinct priorities. The model can be easily changed to account for different assumptions.

4.4.1: Scheduling Overhead

When a task requests service, the control structure must compare the priority of the task with the priority of the currently executing task. If the priority of the current task is higher, then new request must be queued in some manner. When any task completes execution, the control structure must select a new task to execute. Also, switching the processor between tasks will generally involve setting up some processor registers. However, all of these actions will occur for every instance of a task requesting service, so these overhead costs can be included in the maximum CPU time used by task $I = t_I$. The basic algorithm of finding a task which can be assigned the worse priority while still satisfying (4-6) is still correct.

4.4.2: Non-preemptive Control Structures

If the currently executing task always runs to completion before a new task is run, then the latency specification for a task must be large enough to allow for any task with worse priority to execute as well as the CPU time used by tasks with better priority. Thus, (4-6) becomes:

$$l_i \geq t_i + \sum_{j=1}^{i-1} C_j(l_i) + \max_{j=i+1}^{n} (t_j) \qquad (4\text{-}12)$$

Again, the assignment algorithm does not require any changes. This is obvious if the algorithm finds a valid assignment of priorities. Increasing the priority of some task relative to task i moves a task into the summation term in equation (4-12). Since $C_j(t)$ is greater than or equal to t_j, making this change can only increase the right hand side of (4-12).

4.4.3: Non-Distinct Priorities

For various reasons it may be desirable to assign several tasks identical priorities. For example, the computer hardware may only support a limited number of interrupt priorities. Since the control structure is free to execute any of the requested tasks having the highest priority, all tasks having the same priority as task i must be treated as if they had higher priorities when checking the latency specifications. This assumes that the control structure only executes task i when all other requested tasks have priorities strictly worse than p_i.

However, this also makes the often unrealistic assumption that a task can be preempted by a task with equal priority. If this is not the case it is necessary to

simulate the control structure on the worst case sequence of requests. It is not sufficient to treat these tasks as if they had lower priority but are not preemptible since a pair of tasks can make a sequence of requests so that one of them requests service again while the other is being executed. Therefore, the first task can be executed twice while task *i* is waiting for service although task *i* is never preempted.

4.5: Applications to the Control Structure Problem

Verifying the real-time performance of a static priority scheduler on more complex task structures is a straightforward extension of the verification for independent tasks. A latency specification l_i is satisfied if and only if all blocks in the constraint path can always be executed during any interval of duration l_i. It becomes slightly more complex to compute the amount of CPU time used by higher priority tasks since some tasks (blocks) will not be runnable when other tasks are requested.

4.5.1: Chains of Independent Tasks

If no block appears in more than one constraint path, the constraint paths can be treated as independent tasks. A task will never be interrupted by a request of a predecessor if the real-time specifications are met since the period between requests is not less than the deadline for any one request.

The priority assignment problem would be very much more difficult if it were necessary to consider assigning different priorities to individual blocks in a chain.

However, it does not make sense to assign lower priorities to some blocks in the constraint path, since it makes no difference where in the chain higher priority tasks are allowed to interrupt. Therefore, all the tasks in the chain can be assigned the same priority as the task in the chain with the least priority.

In the presence of overhead it is more efficient to create one 'super-task' that executes all the blocks consecutively rather than incurring the overhead of a request for each block in the chain. However, if the control structure is non-preemptive it may be necessary to create several smaller 'super-tasks' to reduce the amount of time that must be spent waiting for low priority tasks to complete. Deciding how many tasks to create and how large to make them could be made on the basis of how much CPU time needs to be freed up in order to find a task to assign the currently worst priority.

4.5.2: More Complex Task Relations

There are fundamentally two ways different constraint paths can have a common block: the common block can have more than one successor or it can have more than one predecessor. We will first consider the simplest example of each type of interdependent constraints.

Consider a block diagram in which block *A* has successors *B* and *C*. The constraint paths for this diagram are *AB* and *AC*. Since a request for *A* will always cause requests for both *B* and *C*, $B_{AB} = B_{AC}$. Therefore, neither *B* nor *C* will be interrupted by requests for *A* as long as the real-time specifications are met.

Now, if $D_{...} > D_{...}$ then the sequence of blocks executed whenever *A* is requested