MIT/LCS/TR-241

REPRESENTATION AND ANALYSIS OF REAL-TIME CONTROL STRUCTURES

Rowland F. Archer, Jr.

August 1980

This blank page was inserted to preserve pagination.

REPRESENTATION AND ANALYSIS OF REAL-TIME CONTROL STRUCTURES

ROWLAND FRANK ARCHER, JR.

©Massachusetts Institute of Technology 1978

September, 1978

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.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

REPRESENTATION AND ANALYSIS OF REAL-TIME CONTROL STRUCTURES

by
ROWLAND FRANK ARCHER, JR.

Submitted to the Department of Electrical Engineering
and Computer Science
on August 18, 1978 in partial fulfillment of the requirements
for the Degree of Master of Science

ABSTRACT

A new notation is introduced for representing real-time scheduling at the task and event level. These schedules are called control structures. The primary constructs included which direct the flow of control are sequencing, iteration, and preemption. Additional notation allows the representation of interrupt masking, task termination by external events, task restart as well as resumption from the point of preemption and codestripping. Algorithms are given for finding the preemption structure of a given control structure in the notation.

The types of representable control structures are classified by the topology of their Control Flow Graphs. It is shown that although branching is allowed in the preemption structure, a tree-shaped preemption structure cannot be represented. Both partial and total orderings of tasks and interrupt priorities are supported, however.

A terminology for describing real-time properties of control structures is developed, and it is seen that without certain assumptions about task execution times and event timings, conclusions cannot be drawn regarding real-time performance of a control structure. A series of algorithms is presented which make use of these assumptions, and find values for task execution times in the presence of preemption. The algorithms can analyze control structures containing the principal control features; suggestions are given for further development of algorithms which can analyze any representable control structure.

Thesis Supervisor: Stephen A. Ward
Assistant Professor of Electrical
Engineering and Computer Science

Key words and phrases: Real-time, control structure,
control flow graph, scheduling, interrupts, latency, codestripping.

Acknowledgements

Primary thanks are due to Steve Ward for the germinal idea, and his guidance in helping me to develop it. His resourcefulness was responsible time and again for keeping this research in motion.

Tom Teixeira's work in this area was also invaluable, especially for his careful and rigorous definitions of real-time properties of control structures.

The excellent systems programming support of the DSSR group has provided an exceptionally hospitable environment in which to program and produce this document.

My wife Lizbeth deserves mention for her encouragement and for doing more than her share so that my attention could stay focused on this investigation.

TABLE OF CONTENTS

1: Introduction 8.

1.1: Related Research 9.
1.2: Objectives 11.
1.3: Outline of the Thesis 14.

2: A Notation for Real-time Control Structures 15.

2.1: Introduction 15.
2.2: The Basic Control Structure 15.
2.3: Flow of Control 17.
2.4: Closed Control Structures 18.
2.5: Iteration 19.
2.6: Preemption 19.
2.6.1: Preemptible Control Structures 19.
2.6.2: Multiple Priority Level Control Structures 21.
2.6.3: Occurrence of Events 25.
2.6.4: Substructure at a Single Priority Level 26.
2.6.5: Determining the Interrupt Structure 27.
2.7: Non-preemptible Tasks 31.
2.8: Stopping the Flow of Control 32.
2.8.1: Breaks in Event Coupled Lists 33.
2.9: External Termination of a Control Structure 34.
2.10: Return of Control to a Preempted Task 35.
2.10.1: Conditional Restart of a Control Structure 38.
2.11: Codestripping 39.

3: Representational Power of the Notation 41.

3.1: Introduction 41.
3.2: Control Flow Graphs 41.
3.2.1: Priority Levels 43.
3.3: Interrupt Driven Control Structures 43.
3.3.1: Globally Cyclic Control Structures 44.
3.3.2: Acyclic Control Structures 47.
3.3.2.1: Branched Control Structures 48.
3.3.3: Locally Cyclic Control Structures 50.
3.3.3.1: Dynamically Decreasing the Range of LC 50.
3.3.3.2: External Termination of Local Cycles 51.
3.3.3.3: Restrictions on Local Cycles 52.
3.4: CFGs at the Task Level 53.

4: Real-time Properties of Control Structures 55.

4.1: Introduction 55.
4.2: Weights of Task Identifiers 58.
4.3: Properties of Event Variables 59.

5: Algorithms 62.

5.1: Introduction 62.
5.2: Latencies in the Absence of Preemption 63.
5.3: Latencies of Constraints in Cyclic Control Structures 67.
5.4: Latencies of Constraints in Preemptible Control Structures 70.
5.4.1: Definitions and General Approach 72.
5.4.2: Finding Infinite Latencies 74.
5.4.3: Delay Due to Preemption 78.
5.4.4: Applications of PTIME 81.
5.4.5: Adding Phase Relationships to PTIME 83.
5.4.6: Task Execution Time with Preemption at Priorities > 0 85.
5.4.7: Latencies for Constraints at Priorities > 0 88.
5.5: Special Cases and Extensions 97.
5.5.1: External Termination 98.
5.5.2: Restart Control Structures 99.
5.5.3: Codestripping 100.
5.5.4: Non-Preemptible Tasks 100.
5.5.5: Stopping the Flow of Control 101.
5.5.6: Constraints at More than One Priority Level 101.
5.5.7: Finite Event Queues 101.

6: Conclusions and Directions for Future Research 102.

Appendix A: Summary of BNF for Real-time Control Structures 108.

References 110.

LIST OF FIGURES

2.1. Syntax for task identifiers. 16.
2.2. Syntax for closed control structures. 18.
2.3. Syntax for preemptible control structures. 20.
2.4. Computing the matrix L 23.
2.5. Initiating events for Example 2.1. 23.
2.6. The I matrix for Example 2.1. 24.
2.7. I+ for Example 2.1. 24.
2.8. Preemption structure for Example 2.1. 25.
2.9. Syntax for event coupled preemptible control structures. 26.
2.10. Initiating events for Example 2.2. 28.
2.11. Computing I for cs's containing event coupled lists. 29.
2.12. Preemption structure for Example 2.2. 30.
2.13. Syntax for non-preemptible tasks. 31.
2.14. Examples of processor idling. 33.

3.1. CFG for (((A B)/e1)C)*. 42.
3.2. CFG for Example 3.1. 45.
3.3. CFG for Example 3.2. 46.
3.4. CFG for the control structure ((A/e1)(B C)D). 47.
3.5. CFG for the control structure (A/(e1:(B/(e2:C|e3:D))|e4:E)). 48.
3.6. A tree-shaped CFG, for (A/(e1:B|e2:C)). 48.
3.7. A CFG which has no corresponding control structure. 49.
3.8. A representable tree-shaped CFG. 50.
3.9. CFG for Example 3.4. 51.
3.10. CFG with an illegal back arc. 52.

5.1. Breakdown of a finite task list into sublists. 64.
5.2. Preemption structure for (5.9). 76.
5.3. Partitioning the events of (5.9). 76.
5.4. Partial execution of a critical window α_m. 91.
5.5. Partial execution of β_1. 94.

1: Introduction

In an article entitled "Toward a discipline of real-time programming" [Wirth 77b], Niklaus Wirth has divided programming into three categories based on the increasing complexity of validating their programs:

1. Sequential programming
2. Multiprogramming
3. Real-time programming

In a real-time system, a program may be attempting to control or to react to certain external processes which cannot be forced to cooperate with programmed processes through use of a synchronization primitive such as a semaphore [Dijkstra 68] or a monitor [Hoare 74]. In order to coordinate itself with these external, non-programmable processes, the real-time program must know something about its own execution speed. Thus its correctness will be dependent on the speed of the processor on which it is run; but this is not a property of the program itself; Wirth identifies this as the essential distinguishing feature of real-time programming.

This thesis does not directly address the issue of validating real-time programs. Instead, it deals with the representation of schedules for real-time programs called *control structures*, and some aspects of measuring real-time properties of the resulting control structures. In the sense that knowledge of these real-time properties may be a prerequisite for validation of a real-time program, the work presented here does represent a contribution to one aspect of the validation problem.

1.1: Related Research

Most of the previous studies in the field of real-time programming have been centered on one of two major areas, the design of languages for real-time programming, and scheduling to meet real-time deadlines.

The development of languages for real-time programming can be split between two approaches; the extension of existing languages [Benson 67; Freiburghouse 77; Ormicki 77; Phillips 76; Wirth 77a], and the creation of entirely new languages tailored to the requirements of real-time programs [Hennessy 75; Kieburtz 75; Schoeffler 70]. The essence of these efforts has been to provide some interface between the real-time program and the scheduling of itself and other programs, either through access to the processor's interrupt system, clocks and/or timers, or by influencing the processor's scheduling routines. Such features provide only a low level capability for determining a process' real-time behavior; in some cases it may be possible to think of all the timing interactions that could impact on the correctness of a real-time system, but the burden of doing so has usually fallen most heavily (and often totally) on the programmer. Decisions as basic as assigning priorities to different tasks must typically be made by manual analysis, in the hope that nothing has been overlooked. As the size of the system increases, the complexity of the problem grows as well, until manual analysis becomes extremely tedious and error-prone, if not impossible.

Ideally, a programmer could submit his real-time response requirements along with his programs, and either have them scheduled appropriately, or be told that his requirements cannot be met by that particular system. Some systems (such as the CONSORT system of the Domain Specific Systems Research group at MIT) have

been developed which can do this for a limited class of programs, but to the author's knowledge no one has yet created a system to do this in general. However, considerable research has been done on scheduling tasks in the presence of hard real-time deadlines.

Most of the significant results obtained have been based on restricting attention to limited classes of control structure types. For example, in a multiprocessor environment where there is a partial ordering of tasks but no iteration outside of tasks, [Manacher 67] gives an algorithm which will construct near-optimal task lists (execution orderings) for almost any combination of task run times and deadlines. If the schedule is full to capacity with tasks whose completion times are guaranteed, his strategy allows the system to take on additional unguaranteed tasks without affecting the guaranteed status of those tasks already scheduled. His scheme does not consider the effects of preemption, however. Serlin [Serlin 72] and Liu and Layland [Liu 73] have independently studied the problem of scheduling tasks which are iterated but have no relative orderings. Serlin gives scheduling algorithms based on fixed priorities, time slicing, and relative urgency. The last is a dynamic priority scheme, where the processor re-evaluates the priorities of each task at every interrupt, and selects for execution the one with the earliest deadline. This method is shown to produce a schedule which meets real-time deadlines if any schedule will, but Serlin's analysis neglects the overhead of context switching.

A different approach is taken by [Hennessy 75; Kieburtz 75] in their microprocessor language TOMAL; instead of using an interrupt system, they have a compiler insert calls to a task control monitor (which is created along with the compila-

tion of a set of programs) at specific points in the compiled code. This provides assurance that the task control monitor will get control within a finite and bounded interval, after each *codestrip*, as the code between monitor calls is named. This is similar to a time slicing system which allocates execution time in fixed amounts to each task, but the time slices are synchronized with program execution. The length of the codestrip is determined by the response time requirements of the task, and the compiler can determine whether the programmer supplied requirements are in conflict. The notation given in this thesis has the capability of representing codestrips.

A work which is related to the present one and in fact complementary is that of Teixeira [Teixeira 78]. Much of the terminology used here was developed there, particularly that of Chapter 5, where algorithms for measuring real-time properties are developed. Teixeira also used the regular expression notation of Chapter 2 to denote sequencing and iteration of tasks. His study centers, however, on finding schedules to meet real-time constraints; the orientation of the present work is described in the following section.

1.2: Objectives

The principal goal of this research is twofold; to develop a convenient representation for real-time control structures, and to demonstrate how such a representation is useful as a basis for analyzing real-time properties for specific control structures.

The representation as developed models control structures at the task and in-

terrupt level; the tasks are assumed to be self-contained program units whose execution time is bounded, and interrupts are represented as occurrences of event variables. The event variables could be used to represent any event however, which might be synchronous or asynchronous with respect to the executing task. The notation can represent total and partial orderings among its tasks, and iteration of tasks at a single priority level or across several priority levels. As well as representing conventional single and multi-level interrupt structures, the control structures given here can represent several unconventional preemption structures, including branched structures where each branch has an individual preemption structure which may itself be branched.

As well as representing this basic framework, the capability is provided to represent:

1. Codestripping as previously described.

2. External termination of a task or group of tasks by an event occurrence (as opposed to temporarily preempting them).

3. Indication that a task or group of tasks is not preemptible by a set of events.

4. The choice between restarting a preempted task or group of tasks from the beginning vs. resuming execution from the point of interruption.

Thus a rather general notation is given, which in addition represents all of this information rather compactly. The notation may be used in any application where it is necessary to communicate something about a control structure of this sort, be it human to human, human to machine, or machine to machine. In the second case, the specific applications in mind are representation of a control structure for

analysis, and for describing to a real-time system what sort of control structure it should establish for a set of tasks with real-time constraints. In this vein, the notation is quite independent of machine architecture, and thus a subset of the language can be chosen for a target machine which supports the control features included therein.

This leads into the second goal of the investigation, which is to demonstrate how algorithms can be developed which ascertain real-time properties for control structures of the language. There are several time intervals which are probably of common interest to a large segment of users of real-time programs, such as:

1. The maximum delay between the occurrence of an event and the initiation of its program.

2. The maximum time required to execute a set of tasks at a given priority, with preemption.

3. The maximum time that may elapse without there being an execution of a given set of tasks.

This is not intended to be an exhaustive survey of real-time properties, but rather an introduction to the usage of the notation as the foundation for such analysis. Indeed, it is likely that each real-time system has its own special requirements and characteristics; it is hoped that an appropriate subset of the language can be chosen to model those characteristics, and algorithms developed which are suited to an application's special needs. In addition, many applications will have natural restrictions which lead to simpler algorithms; it is with intent of illustrating this point that several special case algorithms are developed.

1.3: Outline of the Thesis

The next chapter presents a context free grammar for the control structure language, as well as giving the semantics for each construct. Sequencing, iteration and preemption are the principal features, with extensions added as described in Section 1.2. Methods of determining the overall preemption structure of a control structure are also presented.

Having introduced the notation, Chapter 3 presents the concept of a Control Flow Graph (CFG) [Allen 76; Fosdick 76], which gives a graphic representation for the paths of control flow dictated by a given control structure. A definition of absolute priority levels is derived from a control structure's CFG representation. Then a classification of control structure types representable by the notation is given, based on the topology of their CFG's. In addition, some types of control structures which are *not* representable are described.

A terminology for real-time properties of control structures is developed in Chapter 4; the requirements for knowing certain things about event timings in advance is also discussed here.

This leads into Chapter 5, where a hierarchical series of algorithms is presented which are designed to find the worst cases for some of the real-time properties of increasingly complicated classes of control structures. The most general algorithm given is applicable to the set of control structures which includes the basic framework of sequencing, iteration and preemption. The types of modifications which would be required to analyze any representable control structure are discussed, although detailed algorithms are not given.

2: A Notation for Real-time Control Structures

2.1: Introduction

In this chapter a notation for representing real-time control structures will be developed. The intention is to provide a general analytical tool which will be suitable for representing most of the possible ways to share a processor among the members of a set of tasks. This will include:

1. Sequencing: a total ordering of tasks to be executed.

2. Iteration: cyclic execution of some ordered set of tasks.

3. Preemption: a partial ordering of tasks where the occurrence of an *event* forces termination of execution of the currently running task and starts execution of a new task.

A context-free grammar will be developed to define the syntax of the representation. It is summarized in Appendix A.

2.2: The Basic Control Structure

The real-time system to be represented is modelled as a set of procedures to be run, called *tasks*, a *control structure* which specifies the order (or possible orders) in which the tasks may be run, and a *processor* which executes the tasks according to the scheduling constraints specified by the control structure.

Thus the flow of data between tasks, if there is any, need not be a concern; it is assumed that any execution ordering needed to preserve the intended seman-

tics of the computation (data flow) will be embodied in the control structure. For example, if an output of task A is an input of task B, then the control structure associated with their execution should ensure that task A completes execution before task B begins.

Further, the detailed flow of information and control *within* a task, i.e. among its internal variables and instructions respectively, need not be of concern either. It is only necessary that an upper bound on the execution time of a task be established; this is discussed further in Section 4.2.

A task will be represented by a *task identifier* ("<task id>"), which in most of the examples will be a single capital letter (though it need not be). Figure 2.1 shows the grammar which defines task identifiers.

<letter> ::= A | B | C | ... | Z

<alphanumeric> ::= <letter> | <digit>

<digit> ::= 0 | 1 | 2 | ... | 9

Fig. 2.1. Syntax for task identifiers.

Next to a single task, the simplest thing to represent is the sequencing of two or more tasks which are totally ordered. This is done in the natural way, by listing the task identifiers in the order of execution of their corresponding tasks, separated by blank spaces for parsing. A string of one or more tasks will be called a *basic control structure*, or <basic cs>. Note that it is permissible to list a task id more than once in a <basic cs>, to represent the situation where the corresponding task is executed more than once with zero or more other task executions

sandwiched in between.[1] The syntax is:

<basic cs> ::= <task id> | <basic cs> ␣ <task id>

where "␣" represents the blank space terminal symbol.

The simplest control structure is just a basic control structure:

<control structure> ::= <basic cs>

Thus the grammar given so far is sufficient to represent single task execution and sequenced task execution control structures.

2.3: Flow of Control

It is useful to formalize the notion of control flow with respect to control structure execution. The processor follows the "instructions" supplied by a control structure, doing both "applications-oriented" work (when it is actually executing the statements of a task), and "systems-oriented" work (when it is determining which task to execute next according to the constraints embedded in the control structure). In either case, the actual machine instructions being executed at any time will be associated with a particular symbol in the control structure representation; it will be said that at that time the *locus of control* (abbreviated LC) is at that symbol. For example, in the following control structure:

1. Every occurrence of a task id in a control structure represents a separate instantiation of that task, with its own private state. This is used to model reentrantly coded routines.

A B

when instructions of task A are executing, LC is at A; when instructions of task B are executing, LC is at B.

2.4: Closed Control Structures

It is desirable to introduce parenthesization for the grouping of task id's in the natural way. In particular, this will be needed to indicate the *scope* of the various special symbols which will be used for iteration, preemption, etc. It will also be helpful in constraining the class of legal control structures to exclude nonsensical ones, such as those in which some tasks can never execute, regardless of preemption timing considerations. Parenthesized (sub-)control structures will be called *closed control structures*, and the class will be added to as necessary for additional representational power. At the top level, closed control structures will be included in the set of legal control structures. Figure 2.2 gives the syntax for closed control structures; a syntax is also given for closed control structure lists, which will be needed later to represent more complex control structures.

<control structure> ::= <basic cs> | <closed cs>

<closed cs> ::= (<basic cs>) | (<closed cs> <basic cs>) | (<closed cs list>)

<closed cs list> ::= <closed cs> | <closed cs list> <closed cs>

Fig. 2.2. Syntax for closed control structures.

2.5: Iteration

Most real-time process control applications require the periodic repetition of a certain task or sequence of tasks. Borrowing from the notation of regular expressions, the asterisk is used to indicate a endless repetition of a control structure. Its BNF:

<iterative cs> ::= <basic cs>* | <closed cs>* | <basic cs> <iterative cs>

The use of "*" is most easily explained by examples:

A B* ≡ A(B)* ≡ A B B B B B ...

(A B)* ≡ A B A B A B ...

From a flow of control viewpoint, when LC reaches an asterisk following a right parenthesis, it returns to the matching left parenthesis. If it reaches an asterisk following a task id, it repeats that task.

The final expansion of the top-level definition of control structure is:

<control structure> ::= <basic cs> | <closed cs> | <iterative cs>

2.6: Preemption

2.6.1: Preemptible Control Structures

With the class of control structures defined so far, the only execution sequences possible are those in which the order of task execution is entirely predetermined (static). In many situations, a processor will need to respond to

asynchronous events such as interrupts, which may not occur at predictable times. It may be desirable to have such events trigger the execution of a different part of the control structure than was previously in control. Informally, this will be modelled by placing sub-control structures into the overall control structure in order of non-decreasing priority. Demarcation of the priority levels is achieved by indicating that a control structure is *preemptible*. Figure 2.3 gives the syntax for preemptible control structures. Preemption is initiated by occurrence of a particular event (which may be complex),[1] so an *event variable* is included which stands for the event.

<preemptible cs> ::= <control structure> / <event var>

<event var> ::= e<integer>

<integer> ::= <digit> | <integer> <digit>

<closed cs> ::= (<basic cs>) | (<closed cs> <basic cs>) | (<closed cs list>) |
(<preemptible cs>) | (<closed cs> <preemptible cs>)

Fig. 2.3. Syntax for preemptible control structures.

Consider the following simple example, which models a control structure with a single level of interruption:

$$((A^*/e1)B)^*$$

The interpretation of this control structure is that A runs repeatedly until event e1

1. The event variable itself is not complex, but it may *represent* a complex event.

happens; this initiates B, which executes once; then LC returns to A*.

The next section will describe how more complex control structures are represented (using the above syntax), such as those having multiple levels of interruption.

2.6.2: Multiple Priority Level Control Structures

Informally, event variables lie at the interface between control structures of different priority, the control structure to the left of the "/<event var>" construction having the lower priority. If LC is in the lower priority control structure when the event happens, it will move to the control structure immediately to the right of the event variable.

Thus a control structure with three priority levels might appear as:

((((A B)*/e1)C D)/e2)E)*

The preemption structure (for each event, the tasks which it may preempt) is fairly straightforward here; e1 preempts A or B, e2 preempts A, B, C or D. But the notation is capable of representing more complex control structures, and a method of precisely determining the preemption structure is needed.

The "interrupts" or "preempts" relation is transitive; if e1 interrupts A, initiating C, and C is interruptible by e2, then A is interruptible by e2. Moreover, all tasks of a single basic control structure will run at the same priority level, so basic control structures can be considered as units, rather than examining the preemp-

1. Although a later section will introduce the capability of *masking* specific inter-

tion of individual tasks.[1]

The "interrupts" relation will now be formalized, i.e. it will be established clearly for each event in a control structure which basic control structures it may preempt. The set of tasks which are interruptible by a certain event will be referred to as the *scope* of that event. The "interrupts" relation for a control structure will be represented by a Boolean matrix I with n rows and columns, where n is the number of basic control structures in the control structure being analyzed. A single basic cs is associated with each row i and column i, for $1 \leq i \leq n$. The basic cs associated with row (and column) i will be referred to as "basic cs i."

The first event to the left of each basic cs will be called that basic cs's *initiating event*. If $I[i,j] = 1$, it means that basic cs i runs at a higher priority than basic cs j; in particular, it means that basic cs i's initiating event can preempt basic cs j. The matrix I is computed according to the algorithm given in Figure 2.4.

This matrix specifies which events cause preemptions across the border between adjacent priority levels. Since the "interrupts" relation is transitive, the transitive closure of this initial relation is the complete preemption structure; this specifies, for each event in the control structure, exactly which basic cs's it can preempt. Computing the transitive closure of the relation represented by I is straightforward. Let I^+ be the transitive closure of I. Then $I^+ = I + I^2 + \ldots + I^n$, where + is normal matrix addition. Boolean matrix multiplication is performed like regular matrix multiplication except 'AND' is substituted for 'TIMES' and 'OR' for

---

rupts while a particular task is executing.

Algorithm 2.1:

1. Let n be the number of basic cs's in the control structure. Associate a unique integer from 1 to n with each basic cs.

2. Initialize I to be an $n \times n$ matrix of zeroes.

3. For each basic cs i, do steps 4 and 5.

4. If basic cs i has no initiating event, leave row i of I equal to all zeroes.

5. If basic cs i has an initiating event e, find the control structure immediately preceding the construction "/e." Call this "control structure k." By the syntax of preemptible control structures, control structure k will be either a basic, closed or iterative cs. For each basic cs j in control structure k, set $I[i,j]$ equal to 1.

Fig. 2.4. Computing the matrix I.

'PLUS'.

Consider an example of a control structure which contains preemptible control structures, and which can be used to illustrate the construction of the "interrupts" relation:

Example 2.1. (((((A B)*/e1)C)*/e2)((D/e3)E))*

Notice that this control structure contains four basic control structures, A B, C, D and E. The initiating events for these basic cs's are as specified in Figure 2.5.

Basic CS	Initiating Event	Row/Column of I
A B	none	1
C	e1	2
D	e2	3
E	e3	4

Fig. 2.5. Initiating events for Example 2.1.

The matrix I is formed following Algorithm 2.1, and it appears in Figure 2.6.

1. A B has no initiating event, so row 1 = [0 0 0 0]

2. C's initiating event is e1. The control structure preceding e1 is (A B)*, which contains the basic cs A B. Thus I[2,1] := 1.

3. D's initiating event is e2. The control structure preceding e2 is ((A B)*/e1)C)* which contains the basic cs's A B and C. Thus I[3,1] := 1 and I[3,2] := 1.

4. E's initiating event is e3. The control structure preceding e3 is D. Thus I[4,3] := 1.

I	A B	C	D	E
A B	0	0	0	0
C	1	0	0	0
D	1	1	0	0
E	0	0	1	0

Fig. 2.6. The I matrix for Example 2.1.

Now, to get the overall preemption structure, compute I+, the transitive closure of I, as shown in Figure 2.7.

I+	A B	C	D	E
A B	0	0	0	0
C	1	0	0	0
D	1	1	0	0
E	1	1	1	0

Fig. 2.7. I+ for Example 2.1.

The preemption relations of the control structure are summarized in Figure 2.8.

LC at	Preemptible by	Initiates
A or B	e1	C
A or B	e2	D
A or B	e3	E
C	e2	D
C	e3	E
D	e3	E
E	none	none

Fig. 2.8. Preemption structure for Example 2.1.

2.6.3: Occurrence of Events

The notion of an event "happening" is purposefully left vague; each application of the notation can attach its own meaning. For the purpose at hand it is sufficient to assume that an event variable is like a flag,[1] which gets set when its associated event occurs. The processor checks all the event variables before beginning execution of every instruction. The following informally describes what happens if any flag is found to be set:

1. In the case where LC is to the right of the event variable which has been set, no immediate effect on execution of the currently running task results. The currently running task is of a higher priority than that which is requesting the interrupt.

2. The event variable remains set until such time as LC is to the left of it and in a basic cs which is preemptible by it, at which time it will cause a preemption.

3. If more than one event corresponding to event variables to the right of LC has happened, then the rightmost one represents the highest priority interrupt (request), and LC moves to the right

1. Generally, a queue of requests is associated with a given event variable, so that additional occurrences of the event will be remembered if they occur before the initial occurrence is noted by the processor. By specifying a length for this queue, a system which remembers an arbitrary number of event occurrences can be modelled.

of it (assuming, of course, that LC was within a basic cs preemptible by the event).

4. Completion of the control structure at a given priority "resets" the event variable which triggered its execution; note that this must be done at completion rather than at initiation so that if the control structure is preempted before it completes, then LC will return to it when it is once again the highest priority control structure requesting processor service.

2.6.4: Substructure at a Single Priority Level

A useful extension to the scheme is to provide for arbitrarily many control structures[1] to reside at the same priority level, but to be initiated by different events. During execution of one of these control structures, occurrence of events in the other(s) at the same priority level will have no (preemptive) effect. The principle syntactic change is to allow replacement of an event variable by an *event coupled list*, as shown in Figure 2.9.

<cs> ::= <control structure> / <event list>

<event list> ::= <event var> | (<event coupled list>) |

(<event coupled list>)*

<event coupled list> ::= <event var>: <control structure> |

<event coupled list> '|' <event var>: <control structure>

where '|' means the terminal symbol |.

Fig. 2.9. Syntax for event coupled preemptible control structures.

1. Of arbitrary complexity, e.g. there may be additional local priority structure.

Consider an example:

$$(A^*/(e1: B \mid e2: C))^*$$

Preemption rights are as follows:

LC at	Preemptible by	Initiates
A	e1	B
A	e2	C
B or C	none	none

Execution of B or C continues uninterrupted to termination. Termination of B or C returns LC to A (unless e1 or e2 has happened again).

A slight modification in the position of the terminal '*' leaves the interrupt structure the same but results in different behavior on termination of B or C:

$$(A^*/(e1: B \mid e2: C)^*)$$

The idea here is that once either B or C has been initiated (through occurrence of e1 or e2, respectively), control is never again returned to A. Instead, B and C will be executed every time e1 or e2 occurs.

2.6.5: Determining the Interrupt Structure

Since arbitrary control structures may reside in an event coupled list, it follows that such structures may contain additional events (or event coupled lists) which trigger even more deeply nested control structures.

This ability to nest control structures raises a new semantic issue; what should be the scope of events which are not at the top level in the event coupled list? The choice made here is to let *any* event in an event coupled list have the

same scope external to the event coupled list that an event variable would have if it were substituted for the event coupled list. Consider the following:

Example 2.2. (A/(e1:((B/e2)C)|e3:((D/e4)E)))*

The scope of e1, e2, e3 and e4 external to the event coupled list (e1:((B/e2)C)|e3:((D/e4)E)) is the same as that of e5 in:

(A/e5)

namely, the control structure to the left of the slash in the construction "/(<event coupled list>)".

The initiating events, as shown in Figure 2.10, are determined as before: the first event variable to the left of each basic cs. The *internal* scope of the event variables is somewhat different, though. Events in event coupled lists may not preempt any task in the list which is separated from the event by a "|". Thus in the above example, e3 and e4 may not preempt B or C. Therefore Algorithm 2.1 must be modified to reflect this. Figure 2.11 shows the resulting algorithm.

Basic cs	Initiating event
A	none
B	e1
C	e2
D	e3
E	e4

Fig. 2.10. Initiating events for Example 2.2.

Algorithm 2.2:

1. Let *n* be the number of basic cs's in the control structure under examination. Associate a unique integer from 1 to *n* with each basic cs.

2. Initialize I to be an *n*×*n* matrix of zeroes.

3. For each basic cs *i*, do steps 4 and 5.

4. If basic cs *i* has no initiating event, leave row *i* of I equal to all zeroes.

5. If basic cs *i* has an initiating event *e*, then this event appears in either a "/*e*" construction or a "|*e*" construction.

   a. If *e* appears in a "/*e*" construction, call the control structure immediately preceding "/*e*" "control structure *k*." For each basic cs *j* in control structure *k*, set I[*i*,*j*] equal to 1.

   b. If *e* appears in a "|*e*" construction, then *e* cannot preempt any other basic cs's in the event coupled list of which it is a member. Its scope starts at the control structure to the left of the "/" in the construction "/(<event coupled list>)". This will be the control structure preceding the first unmatched left parenthesis to the left of *e*. Call this "control structure *k*." For each basic cs *j* in control structure *k*, set I[*i*,*j*] equal to 1.

Fig. 2.11. Computing I for cs's containing event coupled lists.

The control structure of Example 2.2 has the following preemption relationships:

LC at	Preemptible by
A	e1, e2, e3, e4
B	e2
C	none
D	e4
E	none

Since two or more tasks may reside at the same priority level, such as B and C above, a natural question arises; what happens if both e1 and e3 occur "simultaneously," at least within the resolution of the interrupt system.[1] Most systems adopt some arbitrary metric to resolve such situations. A typical one is the distance of the interrupting device from the CPU. A similar approach is taken here. If more than one event is found to have occurred at the same priority level, then control is arbitrarily given to the *first* (leftmost) one in the event coupled list.

However, with the addition of event coupled lists, "forks" are introduced into the preemption structure, as shown in Figure 2.12. A diagram such as this is called a *Control Flow Graph*, and will be defined formally and used extensively in the next chapter. For now it is sufficient to note that this diagram "unravels" the preemption structure so that the relative priority levels of each task are displayed. If two or more events happen together, priority is given to the event which initiates the task having higher priority, as was done before. In the above example, if e1 and e4 happen simultaneously control is given first to E (which e4 initiates).

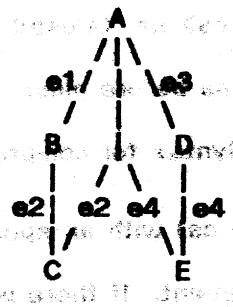

Fig. 2.12. Preemption structure for Example 2.2.

1. Typically the presence of interrupt requests will be checked for once per instruction cycle, so any interrupts happening between two such checks will be indistinguishable as to their ordering in time.

2.7: Non-preemptible Tasks

It is occasionally necessary to perform all or some subset of a control structure's tasks in a non-preemptible mode, even though in the latter case other tasks at that priority level may be preemptible. Simply indicating that a task is non-preemptible is equivalent to saying that the interrupt system is "turned off" while that task is in execution. For generality, the notation allows as an alternative the specification of exactly those events which are not allowed to interrupt the task. Both capabilities are provided with the augmented syntax, shown in Figure 2.13. The scope of the symbol for non-preemptibility extends to closed control structures in the natural way, i.e. every task in the closed cs is non-preemptible.

<basic cs> ::= <task> | <basic cs> || <task>

<task> ::= <task id> | <non-preemptible tid>

<non-preemptible tid> ::= '<task> | '(<ev list>)<task>

<ev list> ::= <event var> | <ev list>,<event var>

<non-preemptible closed cs> ::= '<closed cs> | '(<ev list>)<closed cs>

<closed cs> ::= ...(same as before plus:)... | <non-preemptible closed cs>

Fig. 2.13. Syntax for non-preemptible tasks.

Prefixing a task id (or a closed cs) with an apostrophe (e.g. 'A) indicates that that task is not preemptible by *any* event. If there is an event list after the apostrophe (e.g. '(e1)A), then that task is not preemptible by any event in the event list. Furthermore, it is not preemptible by any event which could lead to preemption by an event in the event list. For example:

((((A*/e1)*(e3)B C)/e2)D/e3)E)*

Here if LC is at B, it is not preemptible by e3 *or* e2, since e2 initiates D which *is* preemptible by e3.

Algorithm 2.2 can still be used to determine the nominal preemption structure for the control structure's set of basic cs's. However, the output of Algorithm 2.2 must then be modified by removing preemptibility relations as specified.

2.8: Stopping the Flow of Control

Although the emphasis has been on how LC moves within a control structure, there may well be times when there is simply no work to be done for the moment. It is worth pointing out how the existing notation indicates this with some examples.

Basically LC will halt when it either:

1. Reaches the "end" of a control structure, and finds no '*', or
2. Reaches a slash ('/') beyond which no events (which are capable of interrupting the control structure to the left of the slash) have occurred.

Several examples are given in Figure 2.14 to clarify this concept; for conciseness, a typical (but not unique) task string which may be generated by each control structure is given. Additional notation should be self-explanatory.

((A*/e1)B)* ----> A A A e1 B A A A e1 B ...

((A/e1)B)* ----> A (wait) e1 B A (wait) e1 B ...

((A*/e1)B) ----> A A A e1 B (halt)

(((A*/e1)B)/e2)* ----> A A A e1 B (wait) e2 A A A ...

Fig. 2.14. Examples of processor idling.

2.8.1: Breaks in Event Coupled Lists

In light of the interpretation given to constructs which result in stopping the flow of control, it will be noted that there is no way to apply iteration to a portion of the control structure which includes all of a lower priority control structure and part of an event coupled list. What is needed is the concept of a *break*, which is essentially a restricted "go to" statement; it directs LC to jump over the rest of the event coupled list to the right parenthesis matching the initial left parenthesis of the event coupled list. Thus it enables the iteration at the end of the event coupled list to be applied to any intermediate part of the list as needed. The syntax for a break is the up-arrow (↑) at the point where the break is desired; it always follows a basic control structure, so it can be incorporated into that BNF:

<basic cs> ::= <task> | <basic cs> ∦ <task> | <basic cs> ↑

As an example, consider the control structure of Example 2.2 modified to include two breaks:

(A/(e1:((B↑/e2)C)|e3:((D↑/e4)E)))*

Now, when LC reaches the end of B or D, it returns to A instead of waiting for e2 or e4, respectively.

2.9: External Termination of a Control Structure

Consider the control structure:

Example 2.3. ((((A^*/e1)B^*)/e2)C)*

Since B^* is *non-terminating* and runs at a higher priority than A^*, A will never be executed again once e1 occurs.[1] There is nothing wrong with this per se, but with the given notation it is not possible to represent the case where occurrence of e2 *aborts* the repetition of B, and returns control to A^* after executing C rather than to B^*.

To do this, the notation must be able to indicate that occurrence of an event terminates execution of a particular control structure, and thus LC does not return to that control structure until its initiating event occurs again. The modified syntax:

<task> ::= <task id> | <non-preemptible tid> | <abort tid>

<abort tid> ::= @<task> | @(<ev list>)<task>

<abort cs> ::= @<closed cs> | @(<ev list>)<closed cs>

<closed cs> ::= ...(same as before plus:)... | <abort cs>

Thus it can be specified that *any* event aborts a task (e.g. @B) or set of tasks

1. Recall that an event "flag," in this case e1, is not turned off until the *end* of the control structure which its occurrence initiates. B^* has no end.

(e.g. @(A B C)) or that any set of events causes termination (e.g. @(e2)B). The event which aborts the task(s) need not be the same as the one which causes preemption in a particular case; execution is terminated as long as the aborting event occurs sometime after preemption and before LC returns to the task.

If the control structure of Example 2.3 is changed to make B an <abort tid>, the desired behavior is obtained:

$$((((A^*/e1)@(e2)B^*)/e2)C)^*$$

Now the string 'A A A e1 B B B e2 C A A A ...' can be generated, where repetition of A and B is for an arbitrary number of times.

2.10: Return of Control to a Preempted Task

There are two distinct choices of what to do when LC returns to a task which was interrupted during its execution: either resume execution from where it left off, or start over again from the beginning of the task. These two strategies will be referred to as *resumption* and *restarting* respectively. Each strategy has its advantages and may be the best choice in different situations. A task which is in-