A SYSTEM FOR COMPUTER MUSIC PERFORMANCE

David P. Anderson
Computer Science Division
EECS Department
University of California, Berkeley
Berkeley, CA 94720

Ron Kuivila
Music Department
Wesleyan University
Middletown, CT 06457

August 21, 1989

ABSTRACT

A *computer music performance system* (CMPS) is a computer system connected to input devices (including musical keyboards or other instruments) and to graphic and audio output devices. A human performer generates *input events* using the input devices. The CMPS responds to these events by computing and performing sequences of *output actions* whose intended timing is determined algorithmically. Because of the need for accurate timing of output actions, the scheduling requirements of a CMPS differ from those of general-purpose or conventional real-time systems.

This paper describes the scheduling facilities of FORMULA, a CMPS used by many musicians. In addition to providing accurate timing of output action sequences, FORMULA provides other basic functions useful in musical applications: 1) per-process *virtual time systems* with independent relationships to real time; 2) process grouping mechanisms and language-level control structures with time-related semantics, and 3) integrated scheduling of tasks (such as compiling and editing) whose real-time constraints are less stringent than those of output action computations.

1. INTRODUCTION

A *computer music performance system* (CMPS) provides a programmable interface between human performers and digitally-controlled output devices. A typical CMPS environment involves several components (see Figure 1):

(1) A human **performer**, while hearing ongoing musical output, generates *performance gestures*. These gestures may be discrete (such as pressing and releasing keys) or continuous (such as varying the pressure on a key).

(2) **Input devices** convert performance gestures into a digital form. Continuous gestures are sampled at a low rate (less than 200 samples/second). Input devices may include a) general-purpose input devices (computer keyboards and pointing devices), b) musical keyboards and other traditional instruments modified to produce digital output, and c) specially-designed musical input devices [37,40].

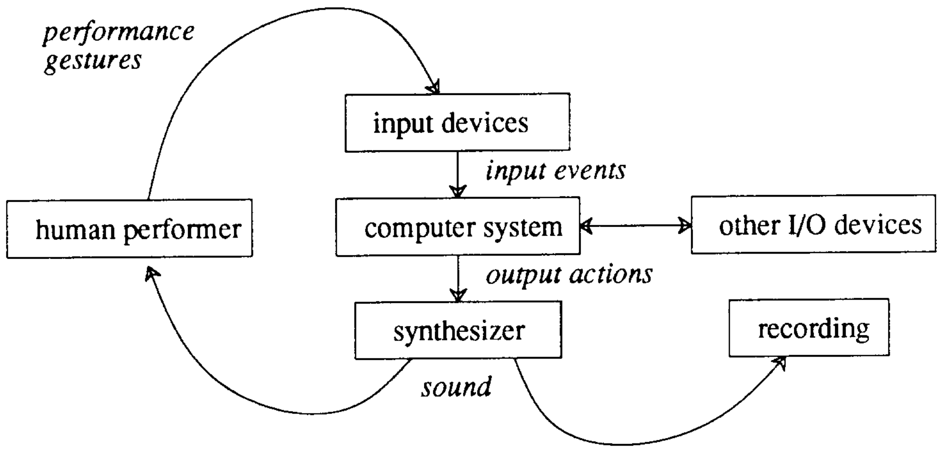

Figure 1: The components of a computer music performance system.

(3) The CMPS itself is a computer system that accepts input from these devices and produces output actions. If these actions are restricted to note start/end commands, the output rate is usually low (less than 100 commands/second). If the CMPS controls continuous note parameters (such as timbre) then the bandwidth may be higher. In addition, the CMPS may be interfaced to other I/O devices such as disks, computer keyboards, and displays.

(4) **Synthesizers** accept commands from the control computer (the MIDI interface standard [16] is a popular means of conveying these commands). The synthesizers generate audio waveforms that are amplified and played back to the performer and audience, and perhaps recorded as well.

The applications of a CMPS include real-time algorithmic composition, automated accompaniment [13, 14, 39], and interactive environments for human performers [9, 40]. To accommodate this range of applications, a CMPS must be programmable. Depending on the application, the programmer has the role of instrument designer, composer, or both. The *system software environment* of a CMPS includes 1) a language by which the musician can program the CMPS and 2) a runtime library and operating system supporting the semantics of the language. Figure 2 shows the major system software components of a CMPS; arrows represent functional dependencies. Programming languages for computer music are described elsewhere [7, 11, 17, 23, 33, 34, 36]. We concentrate here on the operating system level, and in particular the *real-time facility* responsible for scheduling processes and output actions while meeting the stringent timing requirements of music.

FORMULA (Forth Music Language) is a CMPS that is currently being used for instruction and experimentation in several universities, and by many individual musicians and composers. It includes a Forth-based programming language and an operating system supporting the language. This paper describes the structure of FORMULA's operating system. Other aspects of FORMULA are described elsewhere [2-6].

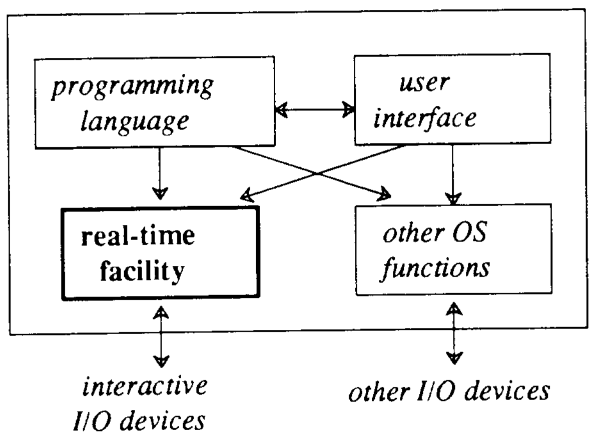

Figure 2: The software structure of a computer music performance system.

The paper is structured as follows: Section 2 explains how the timing of musical output is represented and specified in FORMULA, and Section 3 describes scheduling mechanisms used to increase timing accuracy. Section 4 deals with the advanced programming features of FORMULA, including time deformations, process groups, time control structures, mutual exclusion, and background processes. Section 5 gives implementation details. Section 6 surveys other CMPS designs. Section 7 summarizes the novel aspects of FORMULA and discusses areas of future research.

2. TIMING IN A CMPS

A CMPS responds to *input events* by performing computations and generating *output actions*. For example, a program might sound a note immediately after the performer presses a key on a musical keyboard (in practice, of course, there is a slight delay). We call this a *simple response* because the timing of ouput actions need not be specified.

An input event may also trigger a series of actions that occur over time. These *response sequences* are algorithmically generated: the timing of the actions, and the actions themselves, are dynamically computed by the application program. For example, the response to a key-down event might be a) several delayed repetitions of the key (an echo effect); b) a pre-defined musical phrase or piece, perhaps with some algorithmic variation; c) a variation in the tempo of an ongoing ''accompaniment'' process, to match the tempo of the human performer.

To support multiple simultaneous response sequences, FORMULA provides the abstraction of concurrent processes. Processes have separate stacks, and their state is maintained in activation records on these stacks. A new process is created using[1]

```
create_process(procedure, arguments);
```

The new process executes the given procedure with the given arguments, and exits on return from this procedure.

The system maintains a *time position* for each process. A new process inherits the time position of its parent process. If the process is created in response to external input, its time position is the time when the input was received. A process can advance its time position using

```
time_advance(delay);
```

Delay is expressed in the units of a *virtual time system* not necessarily equal to wall-clock time. Notated time in music is an example of a virtual time system: as tempo fluctuates, time intervals notated as equal are performed with different durations.

Different processes have different virtual time systems, and these systems can run at different rates. This allows different processes to follow distinct tempo changes and *rubato*. For example, a melody played *espressivo* may require tempo changes that do not affect the rhythm of the accompaniment [30].

Virtual time is expressed in integer units. Programs can also specify time intervals in rational (n/m) form. A per-process scaling factor is used to convert rational time intervals to vir-

1 For understandability we use C-like notation throughout. The actual interfaces are in Forth.

tual time. This allows the programmer to use familiar rhythmic units (quarter notes, triplets, and so on).

Although processes may have distinct virtual time systems, these systems all have an unambiguous relationship to the two ''global'' time coordinate systems FORMULA uses. *Real time* (or ''wall clock'' time) is maintained by a periodic clock interrupt. *System time* (ST) differs from real time in two ways. First, ST is related to real time by a scaling factor `global_tempo`, defined as units of ST per unit of real time. Second, if processes fall too far behind schedule, ST stops advancing to allow them to catch up. The degree of lateness tolerated is specified by the parameter `max_lateness`, whose value is in ST units.

Global speed can be changed by modifying `global_tempo`; this is convenient for matching the speed of an external timing source. For example, overdubbing on a multi-track tape deck may require periodically adjusting `global_tempo` to match the speed of a ''click track'' generated by the tape deck.

The virtual time position T of a process corresponds to a particular system time S. However, the system times during which the process has time position T can lie, within certain bounds, ahead of or behind S. This serves two purposes:

- It allows *eventual synchronization* of late processes. For example, a burst of input events may saturate the system, causing the response processes to be late. However, the initial time position of each process is the time of its triggering input event, so subsequent timing is not permanently affected. The response sequence can eventually synchronize with ongoing music, like a musician whose attention has wandered.
- By allowing processes to run ahead of schedule, *action buffering* is possible. This is explained in the next section.

3. SCHEDULING COMPUTATIONS AND ACTIONS

As observed by Loy [24], the timing accuracy requirements for output actions are bounded by the ''motor and psychoacoustic capacities'' of the performer and the listener. The ''simultaneity tolerance'' for human perception of the onset time of a sound depends on its frequency content, and may be as low as a millisecond. Because computations can take significant CPU time, specialized scheduling mechanisms are needed to attain the requisite timing accuracy.

FORMULA uses a technique called *action buffering* to improve timing accuracy. In this approach, action generation is factored into an *action computation* that may require significant CPU time and the performance of an *action routine* that does not. Typically, the action computation determines parameters for the action routine. For example, a computation might determine arguments specifying pitch, waveform and envelope information for an action routine that starts a note on a synthesizer by sending a few bytes on a bus or MIDI channel.

The central idea of action buffering is to allow action computation to run ahead of performance, exploiting the otherwise unused CPU time ''between'' actions. Action performance can preempt action computation, and does not require a context switch to execute. Action buffering can improve the timing accuracy of response sequences that require significant computation (see the Appendix). However, user interaction will suffer a *buffer delay* equal to the amount of time the process has run ahead [19].

3.1. Action-Generating Processes

Action-generating processes produce sequences of output actions. The timing of the actions is a function of time position, which is manipulated by calls to `time_advance()`. An action-generating process does not call action routines directly. Instead it calls

```
schedule_action(procedure, arguments);
```

giving the address of the action routine and the arguments to be passed to it. The scheduler will call the action routine at (or near) the real time corresponding to the process's current time position. If this time position is less than the current ST, the scheduler performs the action immediately. A process may also call

```
    schedule_future_action(delay, procedure, arguments);
```

to schedule actions at times relative to its time position. Without this facility, simple musical concepts would become needlessly difficult to program. For example, a *legato* note sequence requires note release actions to be scheduled after subsequent note attack actions, while in a *staccato* sequence the releases precede the attacks. Future actions make it possible to eliminate these considerations by defining a function

```
    play_note( pitch, volume, duration);
```

that schedules both the attack and the release.

3.2. Scheduling Parameters

In addition to time position, the FORMULA scheduler maintains the following *scheduling parameters* for each process *P*:

(1) The nonnegative `max_delay` determines how far ahead of the current ST *P* is allowed to compute. If *P* advances to a time position such that

$$time\_position > ST + max\_delay$$

then *P* is suspended and becomes *dormant*. It remains dormant until the condition no longer holds, whereupon it becomes runnable. This allows per-process limits to be placed on the buffer delay suffered by user interaction.

(2) `Min_delay` determines *P*'s *deadline* as a function of its time position according to the relation

$$deadline = time\_position - min\_delay$$

At any moment, the runnable process with the earliest deadline has control of the CPU. The running process will be *preempted* if a process with a sooner deadline becomes runnable.

A process inherits its scheduling parameters from its parent process. A process can directly modify its `max_delay` and `min_delay` parameters at any time, but it must call `time_advance(0)` to have these changes take effect (that is, to change its deadline and perhaps become dormant).

What is the practical significance of `max_delay` and `min_delay`? One can observe that a process's computation at a time position T takes place while the current ST lies between $T - max\_delay$ and $T - min\_delay$ (see Figure 3). However, this fact by itself is of little practical use, and we will now explore the question in more depth.

We will say that a process is in *startup state* if it has been recently awakened or created in response to an input event, and has therefore not yet had a chance to compute ahead of real time; otherwise it is in *steady state.*

For steady-state processes, the value of `max_delay` reflects a tradeoff between timing accuracy and buffer delay. If `max_delay` is small, the process will respond quickly to input but may experience action timing errors if system load is heavy or its own computations are long. If `max_delay` is large, the process will be more immune to timing errors, at the expense of an

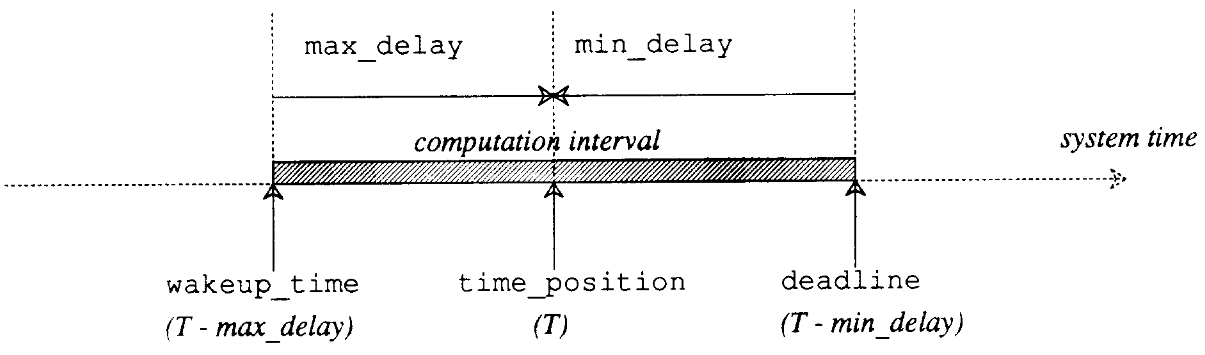

Figure 3: For a given time position, a process's scheduling parameters determine a "window" during which it can execute.

increase in input response time. `Max_delay` can be changed as the role of the process changes during performance. For example, `max_delay` can be temporarily reduced during a period of frequent user interaction with the process.

The `min_delay` parameter can be used to prioritize startup-state processes having approximately the same time position. (The value of `max_delay` is not immediately relevant for such a process, since the current ST will be greater than its time position.) If processes *P* and *Q* have the same time position and *P* has a larger (more positive) `min_delay` than *Q*, then *P* will have an earlier deadline, and will therefore run first (see Figure 4).

For example, suppose that process *P* is awakened in response to key up/down events, while process *Q* is awakened in response to changes in a volume control knob. They both schedule actions with no intervening time advance. In principle, *P* and *Q* should both generate their actions as soon as possible. If, however, the two types of input events occur almost simultaneously, then *P* should execute before *Q* since the timing of its output is more audible. This can be done by giving *P* a larger `min_delay`. (This example also illustrates why `deadline` and

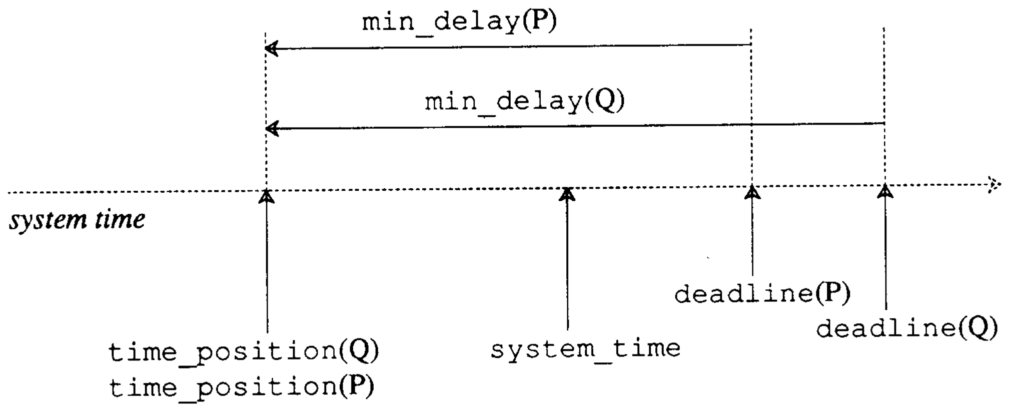

Figure 4: The `min_delay` parameter determines deadline as a function of time position, so it can be used to prioritize processes created or awakened at about the same time.

`time_position` are separate parameters.)

`Min_delay` can also be used to prioritize steady-state processes. Suppose `schedule_future_action()` is used to schedule actions *before* the caller's time position. A process *P* positioned at a particular beat could use this feature to play ''grace notes'' occurring slightly before the beat. *P* should be given priority over other processes at the same time position that do not use this feature by increasing its `min_delay`. *P*'s `max_delay` should also be larger than the ''grace period'' so that the grace note does not occur late.

3.3. Illustrations and Examples

The major concepts of the FORMULA scheduler (action buffering and scheduling parameters) can be illustrated in several ways. First, the states of an action-generating process are shown in Figure 5. Newly created processes are always runnable. When a runnable process's deadline

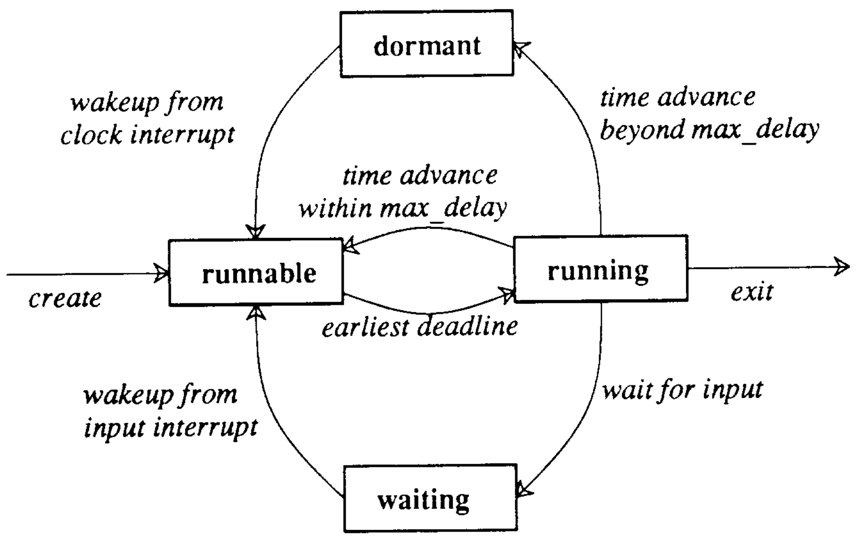
dormant
wakeup from clock interrupt
time advance beyond max_delay
time advance within max_delay
create
runnable
running
exit
earliest deadline
wakeup from input interrupt
wait for input
waiting

Figure 5: States of action-generating processes.

becomes the earliest, it starts to run. It continues until it calls `wait_for_input(device)` or `time_advance(delay)`. The effect of `time_advance(delay)` depends on the magnitude of `delay`. If `delay` advances the new time position beyond the current ST plus `max_delay` the process becomes dormant; otherwise it remains runnable. A dormant process becomes runnable as soon as ST plus `max_delay` catches up with its time position. `Wait_for_input(device)` puts the caller to sleep until there is input from the specified device.

Second, Figure 6 graphs the time position of action computations and performances versus the real time when they occur. The graph is broken into *dormant*, *runnable*, and *late* regions by the lines *time position = real time* and *time position = real time + max_delay.*

The graph depicts the following scheduling scenario. An input interrupt occurs at ST = 1, and its handler creates a process *P* with its `time_position` = 1, `max_delay` = 3, and `min_delay` = 0. The execution of *P* is delayed slightly by processes with earlier deadlines. *P* schedules its first action (A1) late, so the action is performed immediately. *P* then does a `time_advance()` to 5 and schedules an action (A2) for that time. The action is buffered and performed later. *P*'s next time advance, to time 7, exceeds its `max_delay`, and it becomes dormant until ST = 4. Its execution is interrupted at ST = 5 by the performance of action A2.

3.4. Dealing with Lateness

We use the term *system buffer delay* to mean the deadline of the currently running process minus the current ST. Since process deadlines advance independently of ST, the buffer delay varies with time. It is positive when the system is ahead of schedule, negative when it has fallen behind.

The system's response to lateness can be parameterized by imposing a lower bound `max_lateness` on the system buffer delay. When the system buffer delay falls below `max_lateness`, ST stops advancing, and action performance is postponed. When the buffer

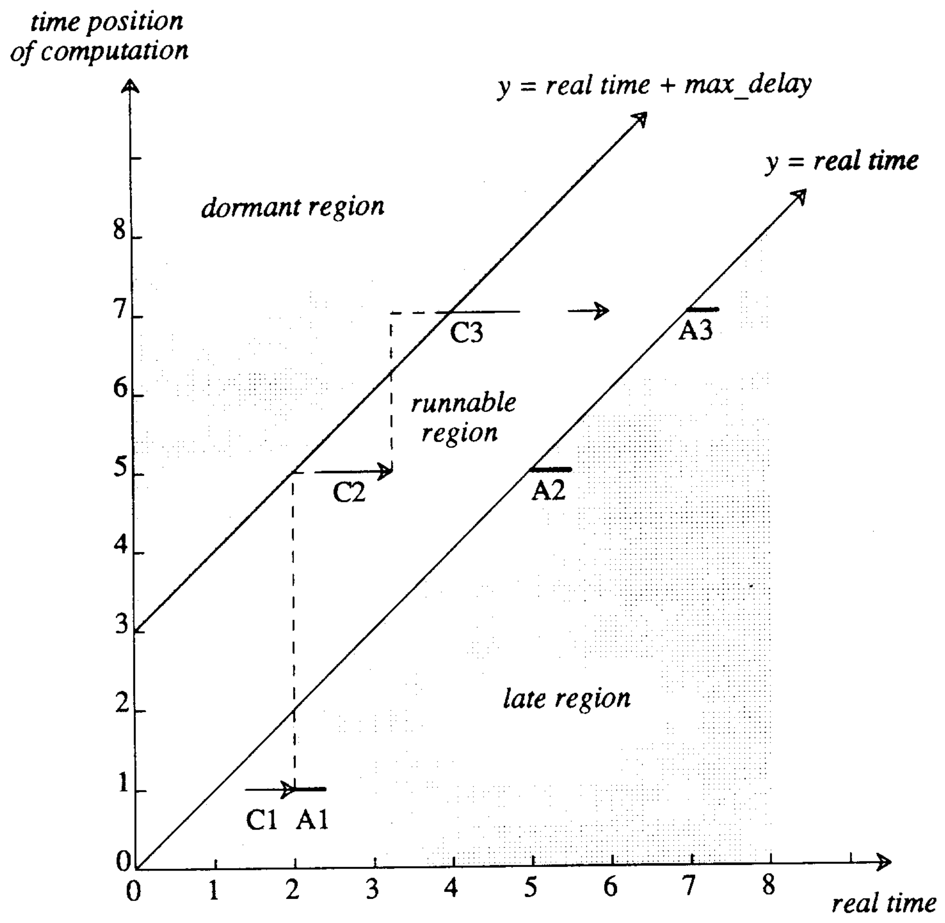

Figure 6: A graph of time position versus real time for a scenario in which a process, responding to an input, executes action computations C1, C2, and C3, which schedule actions with performance routines A1, A2, and A3.

delay again exceeds `max_lateness`, the ST resumes. If ST is stopped for a period *X*, then the entire subsequent schedule of actions is delayed by *X*, and synchronization with the external timing source is lost.

The value of `max_lateness` represents a tradeoff. If `max_lateness` is negative then a limited amount of lateness is tolerated without stopping ST. If `max_lateness` $= -\infty$ then ST is never stopped; this is desirable for applications that must remain synchronized with an external timing source. On the other hand, negative buffer delays can compress the timing of subsequent actions. If the buffer delay assumes a negative value *Y*, then actions whose scheduled performance times differ by less than *Y* may be performed arbitrarily close together. Musically, this distortion of action timing may be less desirable than simply shifting the subsequent action schedule. If this is the case, `max_lateness` should have a nonnegative value.

4. ADVANCED PROGRAMMING FEATURES

4.1. Time Coordinate Systems and Time Deformations

Musical applications require flexible tempo (speed) control. The variable `global_tempo` varies the speed of all processes. FORMULA also provides two mechanisms that allow processes to have distinct tempo variations (see Figure 7).

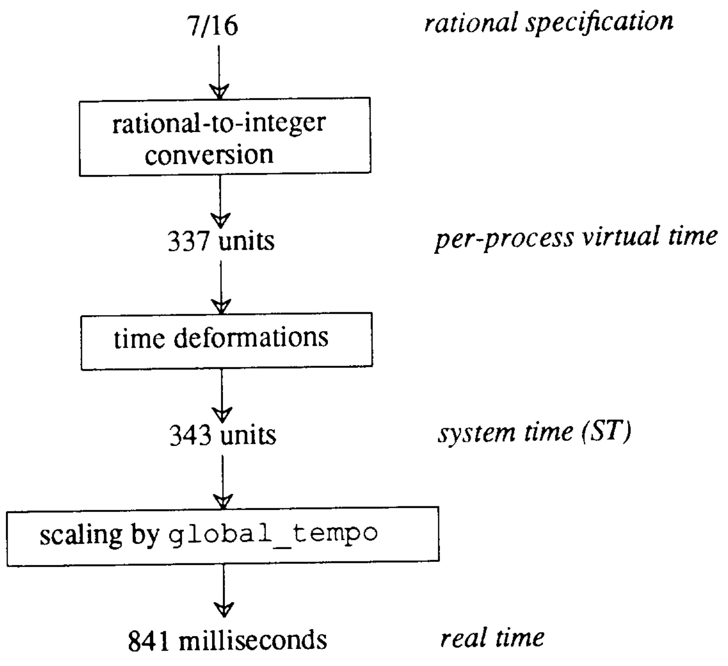

Figure 7: The time coordinate systems used in FORMULA, and the mappings between them.

First, FORMULA provides a function that converts rational numbers into integers with a per-process scaling factor, and accumulates truncation error on a per-process basis. A process's tempo can be subjected to simple (piecewise constant) variation by modifying the scaling factor. FORMULA provides a function `beats_per_minute(n)` that adjusts the scaling factor to provide the given number of (quarter note) beats per minute.

Second, FORMULA provides a *time deformation* (TD) mechanism for more complex tempo variation [6]. A TD defines a *tempo function* that determines the rate of advance of one virtual time system with respect to another. TD definitions are procedures that call *TD primitive* functions, each of which represents a segment of the tempo function. This *procedural concatenation* style of function definition is convenient for musical purposes, since syntactic order corresponds to temporal order (this is not the case with conventional function definitions).

An instance of a TD is implemented as a process that maintains a time position t. A TD process is scheduled as a coroutine that, on each call, accepts an *undeformed* time interval $X > 0$ and returns a *deformed* time interval, obtained by integrating the tempo function from t to $t + X$. X is then added to t. A TD primitive accepts a time advance, integrates its segment of the tempo function over this advance, and performs either a coroutine switch (if the time advance ends within its segment) or a return from procedure (to advance to the next segment). There is also a `pause()` primitive that can be viewed as defining a singular point (with zero width but nonzero integral) in the tempo function. Figure 8 shows a TD viewed as a program, as a tempo function, and as a process.

TDs can be combined in two ways to produce complex time mappings. The *parallel composition* of a set of TDs acts on an input time interval X by supplying X to the TDs in the set, computing their "compression factors" (the ratio of the deformed value of X to X), forming the product of these factors, and returning this product times X. Each action-generating process can have a set of TDs *bound* to it (FORMULA provides language-level mechanisms for defining TDs

(a)

```
for (i=0; i<2; i++) {
    td_segment(0.5, 1.5, 1.0);
}
```

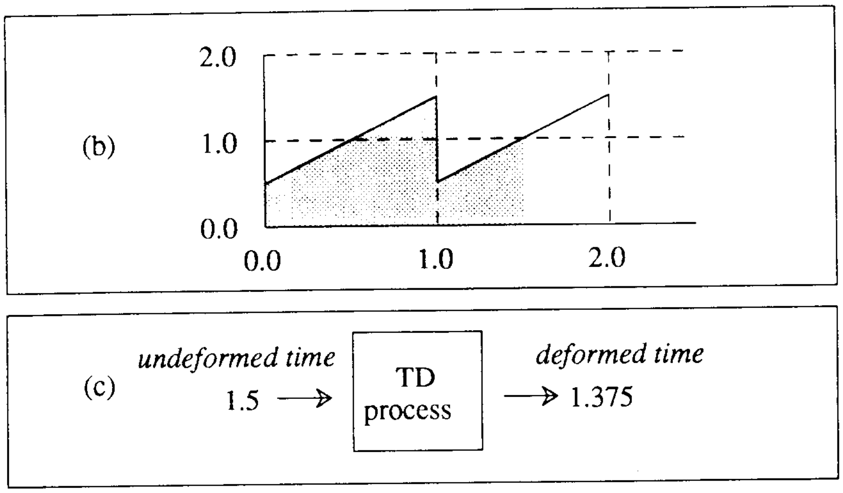

Figure 8: A time deformation (TD) is a program (a) that defines a tempo function (b). An instance of the TD is a process (c) that deforms a time interval by integrating the tempo function over the interval.

and binding them to processes [5]). The time advances of a process *P* are subjected to the parallel composition of the TDs that are bound to *P*. Time deformations can also be combined by ''serial composition'', in which a time interval is deformed by passing it through a series of TDs (or sets of TDs composed in parallel); this is explained below.

4.2. Process Groups

Process grouping mechanisms have been included in some general-purpose operating systems [20]. and are particularly useful in computer music. FORMULA has a facility that allows processes to be collected into *groups* that can be manipulated (suspended, resumed, and aborted) as a unit. Each group can have its own virtual time system and time deformations. A programmer might, for example, group together the set of processes representing a particular instrument or orchestral section.

A FORMULA group is a non-empty set whose members may be either action-generating processes or other groups. We use the term *scheduling object* to mean either a process or a group. Scheduling objects form a set of trees (called *process trees*) under the membership relation. A *top-level* scheduling object is one that is the root of a process tree. The *root set* is the collection of all top-level objects.

A set of TDs can be bound to any scheduling object. This gives each scheduling object its own virtual time system, which is mapped to that of its parent group by the parallel composition of the object's TDs. The mapping from an object's virtual time system to ST is the serial composition of the TD sets between the object and the root set. The virtual time system of the root set is ST.

Processes created using `create_process()` are always members of the same group as the parent process. Their initial time position is inherited from their parent. A group is created by calling

```
    create_group(procedure, arguments);
```

The group is created containing a single new process executing the given procedure with the given arguments. The original calling process sleeps until the last process in the group exits. At that point the group is destroyed and the calling process resumes execution at the terminal time position of the group.

All elements of a process tree share the same `max_delay` and `min_delay` scheduling parameters. If a process in a group waits for external input, its entire process tree is suspended. Therefore process scheduling within a process tree is non-preemptive. One can view a process tree as a single ''activity'' consisting of multiple processes, but governed by a single set of scheduling parameters.

4.3. Time Control Structures

Conventional programming languages have syntactic structures (loops, conditionals, and so on) to control execution flow. These structures serve to combine primitives (simple statements) into compound statements. FORMULA provides a set of *time control structures* whose semantics are based on virtual time usage rather than iteration or conditionals:

```
maxtime(n) statement
mintime(n) statement
minloop(n) statement
```

The `maxtime(n)` structure specifies that the *statement* is to consume at most *n* units of virtual time. When the statement is entered, the upper limit (`time_position` + *n*) is recorded. `Time_advance()` checks whether the advance would exceed this limit, in which case it truncates the advance to reach the limit exactly, restores the call stack to its level at the start of the statement, and transfers control to just beyond the end of the statement. Thus if the `time_advance()` was called from within nested procedure calls, their activation records are removed from the stack. Exiting the statement because of an exceeded limit has no effect on future actions scheduled within the structure. They are executed as scheduled, even if they lie beyond the time limit.

The `mintime(n)` structure specifies that the statement is to be extended by an ''invisible'' time advance, if necessary, so that it consumes at least *n* units of virtual time. When the statement is entered, the lower limit (`time_position` + *n*) is recorded. If, when the end of the statement is reached, the time position is less than this limit, a `time_advance()` is done to reach the limit. The `minloop(n)` structure specifies that the statement is to be iterated, if necessary, so that it consumes at least *n* time units. If the time position is less than the lower limit when the end of the statement is reached, control is transferred back to the start of the statement.

Time control structures can be nested to any depth. This nesting is syntactic; the time intervals specified by the structures need not be nested. The semantics of nesting are that an outer control structure takes precedence over an inner structure. For example,

```
maxtime(40) {
    statement 1
    mintime(30) {
        statement 2
    }
    statement 3
}
```

is meaningful. If *statement 1* consumes more than 10 time units, *statement 3* will not execute.

Time control structures are particularly useful in algorithmic composition, where stochastic musical elements may consume unpredictable amounts of time. The structures allow processes to invoke such elements for fixed or bounded amounts of time, allowing them to be combined more conveniently.

4.4. Mutual Exclusion

FORMULA processes execute in a single address space, sharing both system-level and user-defined data structures. Process scheduling within a process tree is non-preemptive, so operations on data structures shared only by members of a single tree require no synchronization. Scheduling between process trees is preemptive, so any objects shared between trees must be synchronized. Two mutual exclusion mechanisms are used, depending on the type of object:

- If the object has short operations, interrupt-masking can be used. (If the object is shared only among processes, it is sufficient to disable software interrupts; see §4.2) Otherwise, the relevant hardware interrupts must be masked also.
- If the object has long operations (such as calls to a single-threaded window system) then *deadline semaphores* are used. A deadline semaphore is a form of sleep lock. When a process *A* acquires the lock (using the `P()` operation) its deadline and CB are recorded in the lock. If another process *B* preempts *A* and requests the lock, control is transferred to *A* by

temporarily *promoting* its deadline to that of *B*. When *A* releases the lock (using the `V()` operation) its original deadline is restored and it is descheduled. A similar mechanism is used for monitor locks in Swift [10].

4.5. Background Processes

Our discussion of process scheduling has thus far been restricted to action-generating (or *foreground*) processes. These processes perform input response computations. They use `time_advance()` to move between time positions, and consume little CPU time between successive time positions. Tasks such as compiling and editing, however, use significant CPU time and have no precise output timing requirements. To support such tasks, the FORMULA scheduler has a facility called *background processes* (BPs). There are many possible design goals and implementation methods for supporting background processes in a CMPS. We will not enumerate these possibilities, but will simply describe the goals and design of the FORMULA scheduler.

A process may labeled as 1) a foreground process, 2) a background process, or 3) a *latent background process*. A latent BP is one whose characteristics are not known in advance, or can change during execution. For example, a command interpreter process should be a latent BP if it cannot be predicted whether the commands it executes are action-generating. A latent BP is automatically switched between background and foreground, as appropriate, by the scheduler. It becomes a BP whenever its CPU burst exceeds a per-process limit, and it is moved to the foreground when it calls `schedule_action()` or `time_advance()`, and when it is awakened. A BP does not have a meaningful `time_position` parameter. If it makes a call that requires a time position (such as scheduling an action or creating a foreground process), the current ST is used for this purpose.

ST is divided into intervals of a fixed length `window` (perhaps 1 second or so). Each BP has a `slice` parameter, and may be *intrusive* or *non-intrusive*. If a BP is intrusive, its `slice` defines a CPU quota: the scheduler attempts to ensure that the process receives at least that much CPU time within each interval of length `window` (this could be used, for example, for processes that poll I/O devices or that update graphic displays). The scheduler may preempt foreground processes to meet the CPU quota of intrusive BPs. If a BP is not intrusive, it runs only when there are no runnable foreground processes. The `slice` parameter of a process *P* determines the proportion of CPU time that *P* gets in the absence of foreground processes, namely

$$slice_P/(\sum slice_Q)$$

where $slice_Q$ is summed over all background processes.

FORMULA's combination of foreground and background process scheduling mechanisms meets a wide range of single-user CMPS requirements. It supports processes that 1) have precise real-time requirements (foreground processes), 2) require a fixed share of the CPU (intrusive processes), and 3) have no real-time requirements, but should not be starved if possible (non-intrusive processes). The `window` and `slice` parameters allow the programmer to directly the control the proportion of CPU time given to individual BPs, and to adjust the tradeoff between realtime response and BP throughput.

5. IMPLEMENTATION

FORMULA currently runs on a personal computer. The hardware includes an 8 MHz Motorola 68000 CPU, a MIDI interface [16], and a periodic interrupt-generating clock whose period (5 milliseconds) supplies enough temporal resolution for most musical purposes. There is no virtual-memory or floating-point hardware. FORMULA is built on top of Forthmacs [8], a Forth language system. An underlying non-real-time single-process operating system provides file and I/O device access.

5.1. Preemptive Scheduling and Groups

Recall from §3.2 that top-level objects can preempt one another, but that scheduling within a process tree is non-preemptive. This results from the implementation of time deformations as processes. A TD defines the mapping between time systems by a sequence of transformations of undeformed time intervals into deformed intervals. The undeformed intervals must be non-negative because it is not feasible to recover previous states of the TD process. We call this restriction the *monotonic property* of TDs.

This property constrains the scheduling of processes within a group in two ways: 1) processes must be executed in order of their time position within the group, and 2) processes cannot be added to a group with a time position earlier than that of the first (earliest) element of the group.

Scheduling within a tree must therefore be 1) in order of increasing time position and 2) non-preemptive. This in turn implies that all processes within a tree must share `max_delay` and `min_delay` parameters. Top-level groups have scheduling parameters that apply to all their descendants. The parameters have the same semantics as for processes: if a time advance of a top-level group would exceed the current ST plus `max_delay`, then the group becomes dormant until this is no longer true.

In the root set, on the other hand, execution order is determined by deadline rather than time position. When top-level objects have different `max_delay` and `min_delay` parameters their order of execution may not correspond to increasing time position. Consequently, the root set cannot have TDs.

5.2. Scheduling Data Structures

The FORMULA scheduler uses two main data structures (see Figure 9):

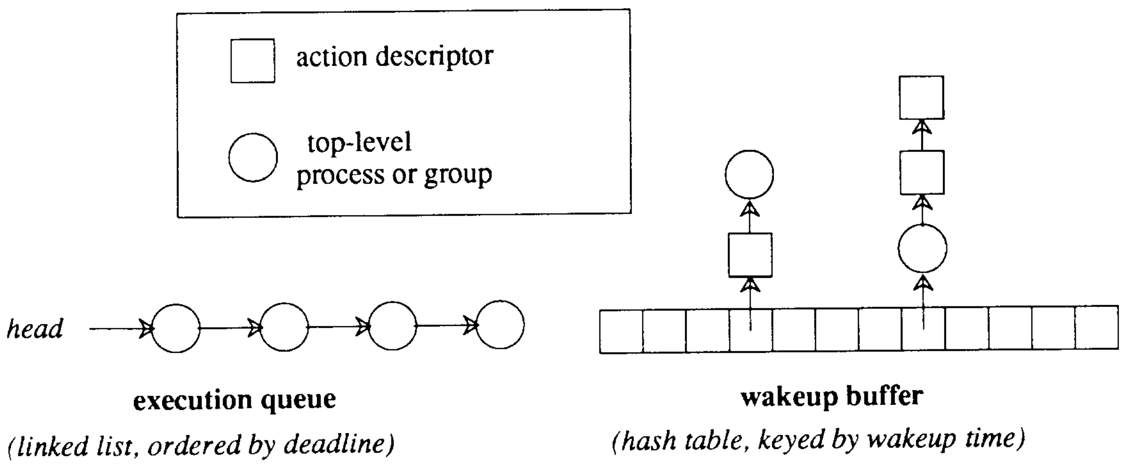
action descriptor
top-level process or group
head
execution queue
(linked list, ordered by deadline)
wakeup buffer
(hash table, keyed by wakeup time)

Figure 9: The FORMULA scheduler uses two main data structures: the *execution queue* and the *wakeup buffer*.

- The *wakeup buffer* stores a) action descriptors and b) dormant scheduling objects (context blocks and group nodes). It is implemented as a hash table with bucket chaining, and entries are hashed by the low order bits of their wakeup times[2]. Each entry has a `wakeup_time` field storing the time (in ST units) when the action is to be performed or the process is to be made executable. With the default value of `global_tempo`, each clock tick corresponds to about one hash table entry.
- The *execution queue* is a list of executable scheduling objects, and is sorted by increasing deadline (in ST units). The executing object is always that with the earliest deadline (the *execution queue head*).

[2] This data structure and various alternatives are discussed by Varghese [38]). In our approach, wakeup buffer entries with different wakeup times may hash to the same bucket. Consequently, the software interrupt has to check the wakeup time of each record in the bucket before processing it. Dannenberg [15] describes a system that eliminates the need for this check by using a hash table only for near-term actions, and using a heap (maintained by a background process) to store distant actions.

Scheduling objects are either processes or groups. Each process is represented by its CB and each group is represented by a *group node* containing a linked list of its elements, sorted by increasing time position.

All scheduling objects have a field `earliest_descendant` that points to the CB of the process in the tree with the earliest time position (if the scheduling object is a process, this field points to the CB of the process itself). A top-level scheduling object has a `preempted` field, a Boolean flag indicating whether the object was preempted. If so, the `preempted-CB` field points to the CB of the process that was executing when preemption occurred (this could be a process other than the earliest descendant; for example, it might be a time deformation process). An example of a process tree, showing various fields of the context blocks and group nodes, is given in Figure 10.

If a scheduling object is preempted, the complete processor state (all registers) is saved on the stack of the preempted process, and the stack pointer is saved at a known location in its CB. The execution queue head's `preempted` field is set, and its `preempted-CB` field is set to the CB of the preempted process. A smaller amount of state (3 registers) is saved for processes that voluntarily suspend execution. When the scheduler does a context switch to an object *X*, it checks the `preempted` flag of *X*. If the flag is true, it follows the `preempted_CB` pointer and restores the full process state. Otherwise it follows the `earliest_descendant` pointer and restores the partial state.

5.3. Implementation of Scheduling Primitives

The implementation of `schedule_action()` is simple. It allocates an action record and copies its arguments to the record. The wakeup time of the action record is set to the maximum of the caller's time position and the current ST, and the action record is inserted in the wakeup buffer.

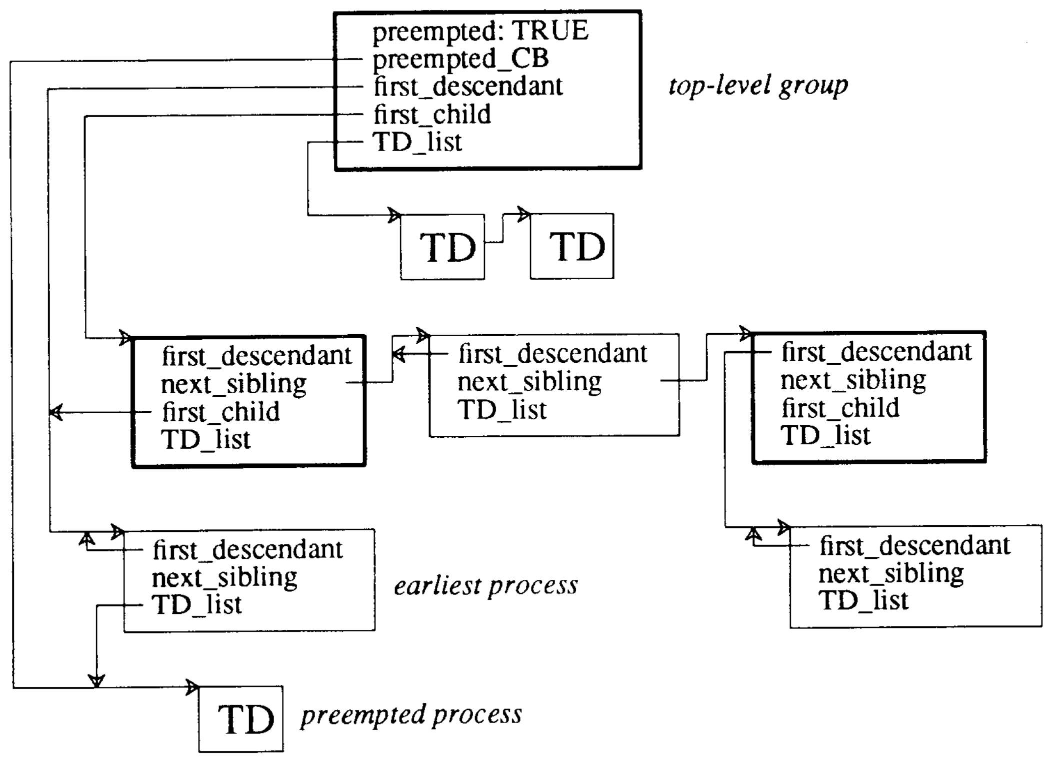

Figure 10: An example of a process tree data structure. The top-level group contains a process and two groups, each of which in turn contains a process. The process tree was preempted while executing in a time deformation (TD) attached to the earliest process. Pointers not drawn are assumed to be null.

Every scheduling object O has two time position fields: $T_I(O)$ is the *internal time position* of the object in its own virtual time system, and $T_E(O)$ is its *external time position* in the system of its parent group. The internal time position of a group is the minimum of the external time positions of its elements. The external time position of a top-level object is ST. $T_I(O)$ is initially 0 and $T_E(O)$ is inherited from the object's creator.

When a process P calls `time_advance(delay)`, `delay` is added to $T_{I(P)}$ and `process_delay(delay, P)` is called. The algorithm for `process_delay(delay, X)` is as follows:

(1) `Delay` is deformed by the parallel composition of the TDs bound to X and added to $T_E(X)$.

(2) If X is not top-level, X is reinserted in the time-ordered list of its parent group G. $T_I(G)$ is advanced by an amount d corresponding to the change from the original $T_E(X)$ to $T_E(Y)$, where Y is the new head of the group. `Earliest-descendant` is propagated from Y to G. Then `process_delay()` is called recursively with arguments d and G.

(3) If X is top-level, software interrupts are masked to protect the execution queue. X is unlinked from the execution queue, and the following fields are updated:

```
time_position += delay;
deadline = time_position - min_delay;
wakeup_time = time_position - max_delay;
```

If `wakeup_time` is greater than the current ST, X is moved to the wakeup buffer; otherwise X is reinserted in the execution queue. In either case, a context switch is done to the new execution queue head. Software interrupts are then unmasked (this happens after the next context switch back to X).

5.4. Action Performance and Preemption

Action performance and process preemption are done jointly by the clock interrupt handler and the software interrupt handler[3]. The division of labor between levels is as follows:

- Action computation is at the process level. Scheduling is preemptive, with priority determined by deadlines.
- Action performance is at the software interrupt level. Scheduling at this level is non-preemptive; action routines are executed in sequence.

[3] A *software interrupt* is requested in software, and has lower priority than any hardware interrupt.

- Clock and I/O interrupts are at the hardware interrupt level. Scheduling is preemptive; interrupt priorities do not change during program execution.

Scheduling between levels is priority-based (in increasing order above) and preemptive.

On each clock interrupt, the handler checks if the earliest process deadline exceeds the current ST by more than `max_lateness`, and returns if so. Otherwise, it adds `global_tempo` to ST. It then checks if any wakeup buffer hash buckets between indices corresponding to the old and new ST are non-empty, and requests a software interrupt if so.

The software interrupt handler processes lists of records from the wakeup buffer. It moves dormant processes to the execution queue and executes action performance routines. It is possible that action routines consume so much CPU time that a second clock interrupt occurs during the software interrupt handler. The software interrupt routine continues to process lists from successive hash buckets of the wakeup buffer until it reaches the current ST. If, on completion of the software interrupt handler, there is a new execution queue head, then the old execution queue head is preempted and a context switch is done. The `preempted` and `preempted-CB` fields of the old process are set (the new process, in this case, never has the `preempted` flag set).

There are two advantages in using the software interrupt handler, instead of hardware interrupt handlers, to perform actions and preemption. First, action performance is done at an interrupt priority below that of all hardware interrupts, so no interrupts are lost because of slow action routines. (In particular, no clock interrupts are lost if action routine execution times exceed a clock period.) Second, since all preemption is done from the software interrupt routine, the problem of preemption within nested hardware interrupts (which in general must be avoided) does not arise.

When a process is awakened by an interrupt routine, it is moved to the wakeup buffer rather than directly to the execution queue. Its wakeup time is assigned so that it will be awakened on the next clock tick. Our reasons for using this approach are:

- Process wakeups to handle input are quantized. Even if several input events happen during a clock period, the handler process is not awakened until the next clock interrupt. This lets the process handle several input events in one CPU burst, reducing context switch overhead. For example, if several MIDI commands arrive in one 5-millisecond clock period, they all get handled in one CPU burst. The slight loss of resolution is insignificant since the clock rate limits output accuracy in any case. Loy suggests the use of input device polling for the same reasons [24]; our solution avoids polling overhead.
- Preemption is done only in the software interrupt handler. This simplifies the structure of the scheduler. It also means that the execution queue data structure is accessed only from process level and from the software interrupt handler, so that it is not necessary to mask hardware interrupts while it is accessed.

5.5. Out-of-Order Action Scheduling

When TDs are used, `schedule_future_action()` knows the *inner* time position, *T*, of the action, but not its position in ST. The ST position cannot be determined until the the process has advanced to *T*, since its TDs may deform the intervening interval.

To solve this problem, each process has a *future action queue* that stores pending actions, sorted by (internal) time position. `Schedule_future_action()` inserts the action record in this queue. The process must assume the time position of each pending action to schedule it. A modified version of `time-advance()` scans the future action queue, subdivides the requested advance as needed. It calls the original `time_advance()` and `schedule_action()` to schedule the future actions at their time positions.

5.6. Implementation of Time Control Structures

Time control structures are implemented as follows. Each action-generating process has two stacks of records (the `max_stack` and `min_stack`) describing the time control structures within which it is executing. Each record contains a time limit, the dynamic nesting level, and possibly a branch address and stack level. Because of recursion, several stack records may refer to a single (syntactic) time control structure. The compiler generates code at the start of each time control structure to create and push the appropriate record.

To implement `maxtime`, `time_advance()` checks the top element on the `max_stack`. If the time limit would be exceeded, a truncated time advance is done, records from the `min_stack` with deeper nesting levels than that of the `max_stack` entry are popped, the process's call stack is restored to the correct level, and control is transferred past the end of the `maxtime` statement.

The compiler generates code at the end of each time control structure as follows: for `max-time`, the code pops an entry from the `max_stack`; for `mintime`, the code pops the `min_stack` and does a time advance if necessary; for `minloop`, the code examines the `min_stack` top and either branches to the start of the statement or pops the `min_stack`.

5.7. Background Process Scheduling: Implementation

BP scheduling is built on top of the deadline scheduler. The runnable BP's are collected into a group (the *BP group*). The *idle process* is always an element of this group; it is non-intrusive and has a `slice` of zero. The BP group is moved between an *infinite* deadline (greater than the deadlines of all action-generating processes) and a *finite* deadline (one that allows it to preempt action-generating processes). When the group's deadline is infinite, a variable `reschedule_delay` stores the delay (in system time) until it is to be promoted to a finite deadline.

BP's are time-sliced while the BP group is running. This time-slicing is among all BP's while the group has infinite deadline, and is only among intrusive BP's while the group has a finite deadline. When all intrusive BP's have received their slices while the group has a finite deadline, it is demoted to its infinite deadline.

The following steps are done on every clock interrupt (context switches are always deferred to the software interrupt routine):

- If the interrupted process is a BP and has a finite deadline, one unit of real time (expressed in ST) is added to its deadline (that is, the deadline of a BP advances as it receives CPU time). If the new deadline exceeds that of a non-background process, the BP is preempted.
- If the interrupted process is an intrusive BP and has an infinite deadline, `reschedule_delay` is incremented. Hence if an intrusive process gets a tick of CPU time at an infinite deadline, the promotion of the BP group to a finite deadline can be delayed by a tick.
- If the interrupted process is a BP that has just finished its slice, a preemptive switch is done to the next BP in cyclic order (or, if the process has a finite deadline, to the next intrusive BP). If all intrusive processes have now received their slices in this window, then the BP group is demoted to an infinite deadline.
- If the interrupted process is a latent BP that has just exceeded its slice, it is moved to the BP group (and possibly preempted).
- If the BP group is at an infinite deadline, `reschedule_delay` is decremented and, if zero, the BP group is rescheduled with a deadline of

  $$ST + window - \sum slice_P$$

  where $slice_P$ is summed over intrusive BP's.

5.8. Performance

Even with a relatively slow processor (an 8 Mhz Motorola 68000), the performance of FORMULA has been sufficient for existing applications. This is due in part to the efficiency of FORMULA's basic process operations. It takes takes a total of 105 microseconds to create a process (allocate and initialize its context block) and delete it (deallocate its context block). A non-preemptive context switch takes 60 microseconds.

A typical FORMULA program involves several note-generat..ıg processes producing a total of 10 to 20 output actions per second, and another 10 or so auxiliary processes such as time deformations. Using `max_delay` values of .5 seconds or so, such a program executes without ever falling behind schedule; all actions are performed at their scheduled 200 Hz clock tick. The system can be made to fall behind schedule only by introducing artificially heavy loads (hundreds of notes per second). In some cases the bottleneck is the limited bandwidth of the MIDI interface. The MIDI input handler can create a new process for each key-down event, with no noticeable ''arpeggiation'' effect when many keys are depressed simultaneously.

6. RELATED WORK

We now survey existing CMPSs, and discuss the use of general-purpose computer systems for computer music performance.

6.1. Discrete Event Simulation

The action-scheduling facility of a CPMS is related to discrete event simulation [21]. Both types of systems offer language-level facilities for algorithmically defining schedules of discrete events. Implementation techniques for discrete event simulation, such as data structures for efficiently storing large numbers of time-ordered records [28], are potentially useful in CPMS design. A major difference between the two types of systems, of course, is that discrete event simulation does not take place in real time.

6.2. Conventional Real-Time Systems

Conventional real-time computer systems [27,29] overlap with FORMULA primarily in their use of deadline process scheduling [22]. Such systems support only simple response to input events. The schedule of output actions is determined by the timing of input events, rather than by a computation. The task of the system is to ensure that the delay of a simple response is small and bounded. These systems also typically offer a *real-time sleep* service that allows a process to sleep for a given real-time delay, or until a given time arrives.

Support for fast simple response and a real-time sleep service are not sufficient to provide accurately-timed response sequences. Action buffering is often necessary for musically acceptable timing. Any system in which the computation of an action begins at its scheduled performance time is susceptible to timing errors that may be musically unacceptable.

6.3. Existing Computer Music Performance Systems

6.3.1. Procedure-Oriented Systems

We use the term *procedure-oriented* to describe a CMPS in which each procedure call takes place at a single time position. Response sequences (which in general may span time) are supported by a service of the form

```
delayed_call(procedure, arguments, time);
```

requesting that the given *procedure* be called at, or soon after, the specified *time*. This approach meshes nicely with object-oriented programming languages; the delayed procedure calls can be cast as *delayed messages* to objects.

The scheduling of calls in procedure-oriented systems may be preemptive or nonpreemptive. The nonpreemptive version (called *intervention scheduling* by Abbott [1]) is used in Moxie [11] and Player [25]. These systems use nonpreemptive scheduling and have no action buffering, so response delay and timing accuracy depend on the execution times of the action-generating computations. System performance may be unacceptable if computations are lengthy.

Object-oriented CMPS's have been built on Smalltalk [18,31]. These are non-preemptive, but some versions incorporate a form of action buffering. The X scheduler [32] is a preemptive scheduler for an object-oriented extension of LISP. It allows the programmer to set upper and lower bounds on the time of the processing of each message. This provides a generalized form of action buffering: an action computation can consist of a chain of messages with wide timing bounds (allowing compute-ahead) while action performance routines are messages with tighter bounds.

6.3.2. Process-Oriented Systems

A *process-oriented* system allows a process to change its time position within a procedure, and within nested procedure calls. A response sequence that spans a time interval can therefore be generated within a single procedure call. PLA [35, 36] and FORMULA [5] are process-based CMPSs.

Compared to process-oriented systems, the procedure-oriented approach is slightly simpler to implement because multiple stacks are not needed. Its main disadvantage is that because processes cannot advance in time, programs must convey state in global variables and arguments to future procedure calls; this can be cumbersome. Process-oriented systems provide a more direct connection between program syntax and musical output.

6.4. General-Purpose Systems and Computer Music Performance

The development of CMPS programs involves tasks such as editing and compiling, and ideally should be supported by a complete programming environment including debuggers, source code management, and so forth. The CMPS designer must decide 1) how to integrate program development tasks with real-time computation, and 2) how to use the software bases of existing programming environments. Three possible approaches are:

- To adapt a general-purpose operating system (such as UNIX[4]) for use as a CMPS. This has the following drawbacks: 1) the process scheduling and synchronization mechanisms of general-purpose systems are not well-suited to computer music performance, and are difficult to modify; 2) the high cost of context switching in systems such as UNIX (because

[4] UNIX is a trademark of Bell Laboratories.

of virtual memory and other overheads) precludes full use of their multiprogramming capabilities.

- To use separate computer systems for development and for performance. For example, the computer music environment developed at CARL [26] uses a special-purpose system for real-time control and a UNIX system for cross-development of programs.
- To extend an existing single-process operating system (typical of current personal computers) to include multiprogramming. The CMPS can then be used for both development and performance. FORMULA takes this approach.

7. CONCLUSION

We have described the real-time requirements of computer music performance systems, and have described a system (FORMULA) that satisfies these requirements. FORMULA provides abstractions that address the needs of computer music performance:

- For generating concurrent action sequences, the basic programming model provides multiple processes with separate logical time positions.
- Processes have *virtual time systems* in which they specify action times. The mapping from virtual time to real time has several levels: 1) rational-to-integer conversion with a per-process scaling factor; 2) time deformations, and 3) global tempo scaling.
- Processes can be collected into *groups* with separate virtual time systems and time deformations. This makes it possible to vary the tempo of a group of processes, and to manipulate a group of processes as a unit.
- *Time control structures* let the programmer bound the amount of virtual time consumed by a section of code. This makes it easier to combine and synchronize musical elements that have unknown or stochastic timing.

- The *background process* facility allow non-time-critical tasks to be scheduled within the same deadline paradigm as other processes.

The implementation of FORMULA uses several novel techniques for supporting the semantics of the programming model. These techniques provide good timing accuracy even on relatively slow hardware.

- The use of *action buffering* serves two purposes. First, it allows accurate action-performance timing even when action computation is time-consuming. Second, in situations where input events are often simultaneous with the actions of existing computations, it makes CPU time available for input handling when it is most needed: near the times when actions occur. This reduces the average latency in input response.
- Using the `max_delay`, `min_delay` and `max_lateness` parameters. the programmer or performer can 1) obtain the optimal balance between timing accuracy and response delay; 2) prioritize both startup-state and steady-state processes, and 3) define a policy for handling lateness.
- The initial (logical) time position of processes created or awakened from input interrupts is the time of the interrupt. In combination with action buffering, this ensures that response sequences are eventually synchronized as closely as possible with one another.

FORMULA shows that it is possible to satisfy the demands of both computer music performance and general system usage (such as program development) in a single computer system. Doing so requires neither more CPU power than that available in current personal computers, nor other special hardware. It is necessary, however, to incorporate special mechanisms (such as action buffering and deadline scheduling) at the lowest levels of the system.

Many areas remain unexplored in the design of FORMULA. The current design deals effectively only with discrete abstractions (such as key up/down actions) and with continuous abstractions sampled at the times of discrete actions (such as time deformations). It would also be useful to offer continuous abstractions (for musical activities such as pitch and amplitude variation within notes) automatically sampled at an appropriate rate. Such an approach is taken in the Arctic language [12]. This capability will greatly increase the computational load, and may require new scheduling techniques.

The current design assumes that action routines use negligible CPU time. If applications were to perform more time-consuming actions (such as graphics output), then long action routines could interfere with the timing accuracy of shorter routines. It might therefore be useful to *prioritize* actions; these priorities could be used to order the execution of action routines, perhaps with the possibility of preemption.

8. ACKNOWLEDGEMENTS

We would like to thank George Homsy, who was involved with several aspects of the FORMULA design and implementation, and Erling Wold, Dale Macdonald and Bruce Holmer, who used early versions of FORMULA and gave valuable feedback.

REFERENCES

1. C. Abbott, ''Intervention Schedules for Real-time Programming'', *IEEE Transactions on Software Engineering SE-10*, 3 (May 1984).
2. D. P. Anderson and R. J. Kuivila, ''A Model of Real-Time Computation for Computer Music'', *Proceedings of the 1986 International Computer Music Conference*, The Hague, 1986, 35-42.
3. D. P. Anderson and R. J. Kuivila, ''Accurately Timed Generation of Discrete Musical Events'', *Computer Music Journal 10*, 3 (1986), 48-56.
4. D. P. Anderson, ''Synthesizer Management Based on Note Priorities'', *Proceedings of the 1987 International Computer Music Conference*, Urbana-Champaign, 1987, 230-237.
5. D. P. Anderson and R. J. Kuivila, ''FORMULA Version 3.4 Reference Manual'', *Unpublished document*, January 1989.
6. D. P. Anderson and R. J. Kuivila, ''Continuous Abstractions for Discrete Event Languages'', *Computer Music Journal*, To appear.
7. L. Boynton, P. Lavoie, Y. Orlarey, C. Rueda and D. Wessel, ''MIDI-LISP: A Lisp-based Music Programming Language'', *Proceedings of the 1986 International Computer Music Conference*, The Hague, 1986, 183-186.
8. M. Bradley, *Forthmacs User's Guide*, Bradley Forthware, P.O. Box 4444, Mountain View, California 94040.
9. X. Chabot, ''User Software for Realtime Input by a Musical Instrument'', *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, 1985.
10. D. D. Clark, ''The Structuring of Systems Using Upcalls'', *Proc. of the 10th ACM Symp. on Operating System Prin.*, Orcas Island, Eastsound, Washington, Dec. 1-4, 1985, 171-180.
11. D. Collinge, ''MOXIE: A Language for Computer Music Performance'', *Proceedings of the 1984 International Computer Music Conference*, Paris, 1984, 217-220.
12. R. B. Dannenberg, ''Arctic: a Functional Language for Real-Time Control'', *Proc. 1984 ACM Symp. on Lisp and Functional Programming*, 1984, 96-103.
13. R. B. Dannenberg and J. Bloch, ''Real-Time Computer Accompaniment of Keyboard Performance'', *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, 1985.
14. R. B. Dannenberg and B. Mont-Reynaud, ''Following an Improvisation in Real Time'', *Proceedings of the 1987 International Computer Music Conference*, Urbana-Champaign, 1987, 241-248.
15. R. B. Dannenberg, ''A Real Time Scheduler/Dispatcher'', *Proceedings of the 1988 International Computer Music Conference*, Cologne, 1988, 239-242.
16. *MIDI Specification 1.0*, International MIDI Association, North Hollywood, California, 1983.
17. K. Jones, ''Real-Time Stochastic Composition and Performance with AMPLE'', *Proceedings of the 1986 International Computer Music Conference*, The Hague, 1986, 309-311.
18. G. Krasner, ''Machine Tongues VIII: the Design of a Smalltalk Music System.'', *Computer Music Journal 4*, 4 (1980), 4-14.

19. R. J. Kuivila and D. P. Anderson, ''Timing Accuracy and Response Time in Interactive Systems'', *Proceedings of the 1986 International Computer Music Conference*, The Hague, 1986, 327-330.

20. T. J. LeBlanc and S. A. Friedberg, ''HPC: A Model of Structure and Change in Distributed Systems'', *IEEE Trans. on Computers C-34*, 12 (Dec. 1985).

21. A. M. C. Leeming, ''A Comparison of some Discrete Event Simulation Languages'', *Simuletter 12*, 1-4 (1981), 9-16.

22. C. L. Liu and J. W. Layland, ''Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment'', *J. ACM 20*, 1 (1973), 47-61.

23. G. Loy and C. Abbott, ''Programming Languages for Computer Music Synthesis, Performance and Composition'', *ACM Computing Surveys 17*, 2 (June 1985), 244-250.

24. G. Loy, ''Designing an Operating Environment for a Realtime Performance Processing System'', *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, 1985, 9-13.

25. G. Loy, ''Player - Extensions to the C Programming Language for Parallel Processing and Music Synthesis Control'', *Unpublished manuscript. CARL software release, CME, UCSD*, 1985.

26. G. Loy, ''Designing a Computer Music Workstation from Musical Imperatives'', *Proceedings of the 1986 International Computer Music Conference*, The Hague, 1986, 375-380.

27. H. Lycklama and D. L. Bayer, ''The MERT Operating System'', *Bell System Tech. J. 57*, 6 (July-Aug. 1978), 2049-2086.

28. J. McCormack and P. Asente, ''Using the X Toolkit or How to Write a Widget'', *Proceedings of the 1988 Summer USENIX Conference*, San Franscisco, CA, June 20-24, 1988, 1-14.

29. A. K. Mok and M. L. Detouzos, ''Multiprocessor Scheduling in a Hard Real-Time Environment'', *Proceedings of the 7th IEEE Texas Conference on Computing Systems*, Houston, Texas, Nov. 1978, 1-12.

30. W. A. Mozart, *Letters of Wolfgang Amadeus Mozart*, Dover, New York, N.Y., 1972. P. 41.

31. S. Pope, ''The Development of an Intelligent Composer's Assistant: Interactive Graphics Tools and Knowledge Representation for Music'', *Proceedings of the 1986 International Computer Music Conference*, The Hague, 1986, 131-144.

32. M. Puckette, ''Interprocess Communication and Timing in Real-Time Computer Music Performance'', *Proceedings of the 1986 International Computer Music Conference*, The Hague, 1986, 43-46.

33. X. Rodet and P. Cointe, ''FORMES: Composition and Scheduling of Processes'', *Computer Music Journal 8*, 3 (1984), 32-50.

34. D. Rosenboom and L. Polansky, ''HMSL: A Real-Time Environment for Formal, Perceptual and Compositional Experimentation'', *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, 1985, 243-250.

35. B. Schottstaedt, ''PLA Reference Manual'', *CCRMA, Stanford University*, June 1983.

36. B. Schottstaedt, ''PLA: a Composer's Idea of a Language'', *Computer Music Journal 7*, 1 (1983), 11-20.

37. J. M. Snell, ''Sensors for Playing Computer Music with Expression'', *Proceedings of the 1983 International Computer Music Conference*, Rochester, New York, 1983.

38. G. Varghese and T. Lauck, ''Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility'', *Proc. of the 11th ACM Symp. on Operating System Prin.*, Austin, Texas, Nov. 8-11, 1987, 25-38.

39. B. Vercoe, ''The Synthetic Performer in the Context of Live Performance'', *Proceedings of the 1984 International Computer Music Conference*, Paris, 1984, 199-200.

40. M. Waisvisz, ''The Hands, a Set of Remote MIDI-Controllers'', *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, 1985, 313-318.

APPENDIX: AN ANALYSIS OF ACTION BUFFERING

Assume that the performer can interact with a process (*e.g.*, by modifying variables) at arbitrary points in its execution. Let us consider the effects of action buffering on the *response delay* of the interaction: the interval from the time of the input to the time when its effects on actions are heard. Let T_I be the real time of an input, P the process's time position at that moment, and T_P the real time corresponding to P. A process cannot, in general, be backed up to an arbitrary previous state efficiently. Hence the input can affect only actions at or beyond T_P. Response delay is therefore at least $T_P - T_I$, *i.e.*, the amount by which the process is "ahead of schedule" [5]. A tradeoff therefore exists: computing farther ahead of schedule increases immunity to timing errors, but reduces interactive responsiveness. Is there a scheduling policy for action computations that is optimal with respect to response delay? We will show that such a policy exists, given complete knowledge of the workload. In the absence of this knowledge, the policy can be approximated; this is essentially what the FORMULA scheduler provides.

Suppose that an action-generating program and a CPU are given. The *load function $L(t)$* is defined as the CPU time required to compute all actions scheduled for time t. For example, a program that generates sampled volume envelopes will have loads at intervals of the sample period, perhaps 10 milliseconds. The size of these loads will be roughly constant from sample to sample. A program that generates samples of a synthesized waveform will have loads at much smaller intervals, perhaps 20 microseconds. Finally, an application that generates note-start and note-end actions will have an irregular load function.

If an action A scheduled for time t is to be performed on time, its computation clearly must be started by $t - L(t)$. If there is another load soon after t, then the computation of A must be started even earlier. This "cascading" effect is captured by the *cumulative load function $C(t)$*,

[5] This bound applies only to interactions with action computations, not to interactions with action performance routines. If action performance routines use user-accessible variables (such as a pitch offset for note-start actions) then changes to these variables are manifested immediately.

defined by

$$C(T) = L(T)$$
$$C(t) = max(0,\ C(t+1) + L(t) - 1) \quad \text{for } t < T$$

where time is quantized so that $L(t)$ is nonzero only for integer values of t, and T is the time of the program's last action (the greatest time for which $L(t) > 0$). Figure 11 shows examples of the load and cumulative load functions for the cases mentioned above. We will show that $C(t)$ is the amount of computational "head start" needed at time t to perform subsequent actions on time, and that it determines lower bounds on response delay.

Let $L(t)$ be given. Assume that time is quantized so that L is integer-valued and is nonzero at integer points $t_1, \ldots, t_n$. Let q_t denote the time unit beginning at time t. A *schedule* is a collection $\{C_1, \ldots, C_n\}$ of sets of time units (C_i is the set of time units during which the action

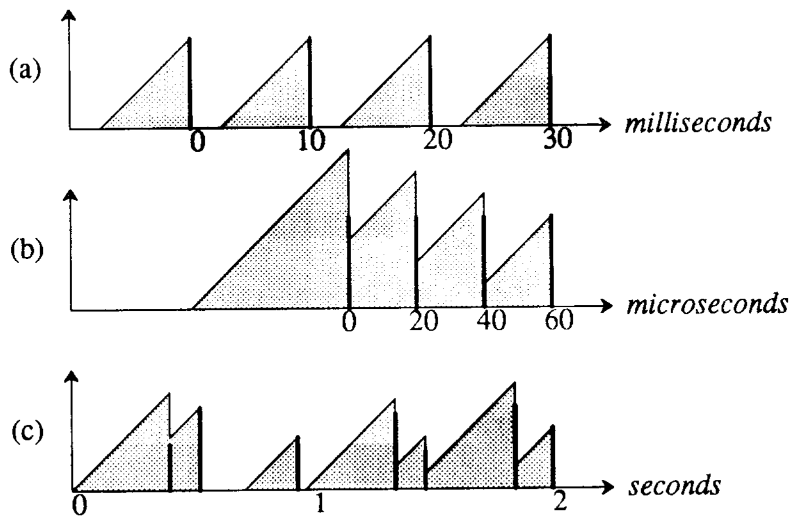

Figure 11: The load function $L(t)$ (bold vertical lines) and cumulative load functions $C(t)$ (shaded regions) for (a) a sampled volume envelope, (b) a sampled audio waveform, and (c) note start/end computations. In example (b), if $L(t)$ continues to repeat indefinitely, $C(t)$ diverges. This implies that the program cannot be computed and performed in real time.

at t_i is being computed). A *legal schedule S* is one that satisfies:

$$|C_i| = L(t_i)$$
$$C_i \cap C_j = \emptyset \text{ for } i \neq j$$
$$q_t \in C_i \Longrightarrow t < t_i.$$

Let C denote $\bigcup C_i$, and let $\underline{C}_i$ denote the least t such that $q_t \in C_i$. The following Lemma is easily shown by induction on the number n of actions:

Lemma: if S is a legal schedule, then for all t_0, $|\{t \leq t_0 \colon q_t \in C\}| \geq C(t_0)$. In other words, at least $C(t_0)$ time units of computation must be done by time t_0.

Given a schedule S, let D_i (the *response delay* of action i) be defined as $t_i - \underline{C}_i$. D_i indicates how far in advance the computation for t_i is begun. A *burst* is an interval $[i, j]$ satisfying $C(t) > 0$ for $t \in [i, j]$, $C(i-1) = 0$, and $C(j+1) = 0$ (the load in Figure 11(c), for example, has three bursts). Each action time t_i is an element of a burst. We now show that, regardless of what schedule is used, $C(t)$ gives per-burst lower bounds on response delay.

Theorem: if S is a legal schedule, then for each burst B and each time $t \in B$, there is an action at time $t_i \geq t$ in B such that $D_i \geq C(t)$.

Proof: induction on the number n of actions in L. The hypothesis is vacuously true for $n = 0$. Assume the hypothesis is true for n, and suppose that L has $n+1$ actions. Let B and t be given. If there are any actions in L strictly before t, consider the schedule in which all such actions (and their computations) are removed. By induction there is an action $A \in B$ satisfying $D_A \geq C(t)$. Now suppose there are no actions strictly before t, and that there is no action $A \in B$ after t with $D_A \geq C(t)$. Then $q_{t-C(t)} \notin C$ because otherwise the action A being computed in $q_{t-C(t)}$ would satisfy $D_A \geq C(t)$. But then at most $t - C(t) - 1$ time units of computation are done before t,