RESPONSE TIMES IN LEVEL STRUCTURED SYSTEMS

by

Paul K. Harter, Jr. *

CU-CS-269-84 July, 1984

Department of Computer Science, University of Colorado,
Boulder, Colorado 80309

* This research was supported by National Science Foundation
Grant # MCS-8216707.

Response Times in Level-Structured Systems

Paul K. Harter, Jr.

Abstract

Real-time programs are among the most critical programs in use today, yet they are also among the worst understood and difficult to verify. Validation of real-time systems is nonetheless extremely important in view of the high costs associated with failure in typical application areas. We present here a method based on an extended temporal logic for deriving response-time properties in complex systems with a level structure based on priority. The method involves a level by level examination of the system, where information distilled from each successive level is used to adjust the results for later levels. The results obtained at each level of the system are static, in that they are not affected by later analyses, which obviates having to consider a complex system as a whole.

Response Times in Level Structured Systems

1. Introduction

The work reported here deals with the problems associated with verifying real-time systems. A real-time system is one which monitors or controls events in an external environment and which therefore must respond within certain hard-time constraints if it is to perform properly. Typical real-time applications include industrial process control, guidance systems and device emulation for hardware testing. The cost of software failure can be tremendous in these applications. Thus, the validity problem for real-time programs is an important one. Unfortunately, it is generally agreed that real-time programs are the most difficult to understand and hence to write, debug, or modify. Further, it is by now almost universally accepted that normal testing and debugging does not ensure working software. Thus, it is felt that it would be best to apply some systematic formal procedure to ensure accuracy. The choice of method is somewhat more highly contested. One promising approach is through the use of formal verification techniques. The application of these techniques to real-time programs, however, is more difficult than to other programs.

To motivate this claim, consider the following. A sequential program involves only a single process, which transforms input data and generates the appropriate output. Programs in this class generally function deterministically and are hence easiest to verify; yet despite considerable progress in the area, large sequential programs are (almost) never formally verified. The difficulties lie largely in finding formal specifications for programs, finding the appropriate loop invariants and manipulating the large complex formulae involved in proofs.

Concurrent programs retain the difficulties involved in the verification of sequential programs and add to them problems of interference among multiple executing processes. In most cases, the strongest assumption allowed about process scheduling is that all processes make "finite progress." As a result, one must make the worst case assumption for concurrent programs, i.e. that the statements of the various processes may execute in any order consistent with the processes themselves. Processes sharing address-spaces may then interleave access to common variables in unpredictable ways. In general, the number of possible interleavings is exponential in the number of processes. Various techniques have been suggested for coping with the interference resulting from interleaving in the context of program verification. These techniques are generally easier to apply where some explicit synchronization is employed to restrict the number of possible interleavings.

Real-time programs deal with events that arise spontaneously in the external environment. These events may be viewed as the product of fictitious asynchronous processes, whose execution speeds cannot be controlled using the explicit synchronization mechanisms provided in concurrent programming languages. Correct functioning of the system, therefore, depends on the ability of the program to keep pace in real time with these processes. As might be imagined, real-time programs have all the problems associated with concurrent programs with the additional problems that go along with a dependence on real time. It is now necessary to consider statement execution times, priority structure, interrupts, and device latency.

Previously reported work [Bernstein 81b] has dealt primarily with proving safety properties of real-time programs, while the work reported here deals with proving a class of liveness properties, which we call response time properties. Using the notation of the above reference, response time properties are those of the form $A \overset{<n}{\rightsquigarrow} B$ ($A \overset{>m}{\rightsquigarrow} B$), which assert that whenever property A is established property B will be (will not be) established before n (until at least m) time units have elapsed. For a real-time program interacting with its environment, it is often necessary to prove just such properties. One may wish to prove that a program polls a device at a certain frequency ($poll \overset{<n}{\rightsquigarrow} next-poll$) in order to make sure no information is lost. One might just as well wish to prove that a program will not poll a device too frequently ($poll \overset{>m}{\rightsquigarrow} next-poll$), as getting the same information twice would prejudice results.

Our previous techniques are, of course, applicable to this problem, however, as the size and complexity of the system increase, the complexity associated with applying these techniques increases dramatically. In this paper we examine a class of systems employing multiple priorities and level structuring and present a technique for deriving approximate results for these systems. We hasten to point out that the error in these results is on the side of conservatism, hence any result proven with these techniques holds in general.

The only other work applicable to level structured systems of which we are aware was reported by Wirth [Wirth 77a]. In that work, Wirth proposed a three-step method for the construction of real-time software, the last of which was a check on the situations involving "races". The proposed checks involved a number of simplifying assumptions on the nature of processes and their interrelationships. Given his assumptions, he gave rules for considering the time lost to a process due to interrupts at higher priorities as an aggregate effect in the average case. The analysis here is similar to that proposed by Wirth. The difference lies in our level-by-level approach, the fact that we consider inter-level procedure calls, and the fact that we derive precise rather than "average case" upper and lower bounds. In addition, our treatment of upper and lower bounds allows us to handle situations in which Wirth's simplifying assumptions do not hold.

In the next section, we review our notation and some of our previous work as required for the understanding of the material to follow. For details, the reader is referred to the report cited above [Bernstein 81b]. In the sequel we define the class of systems under discussion and then present the techniques applicable to that class. Finally, we illustrate these techniques with an example.

2. Previous Work and Notation

In this section we attempt to provide the reader enough background to follow our presentation. We make no attempt here to be precise or complete.

We model the execution of a real-time program as the interleaved execution of the various processes involved. This is a fairly accurate model, particularly for programs running on multiprogrammed machines and has been used in much of the work on the verification of concurrent programs [Owicki 82], [Gabbay 80], [Wolper 81]. Formally, our model includes a set of states and a set of sequences over those states corresponding to the set of possible executions of a program. The primary difference in our model is the inclusion of a time stamp on each state, indicating the time the state was entered. Each non-initial state in a sequence is typically the result of the execution of one "ready" atomic action beginning in the previous state, where an action is "ready" if it is the next action to be executed by a runnable process. To simplify the semantics of our assertion language, execution sequences of terminating programs are extended to infinity by replication of the "last" state. Thus, the execution sequence:

$$s = s_0,\ s_1,\ s_2,\ \cdots$$

represents a program execution that starts in state "s_0," executes an operation leading to state "s_1," then executes another to lead to state "s_2" and so forth. The one exception to the above rule is the case where, in a particular state, there is no ready process. In this instance, the next state will be the result of a process suddenly becoming ready due to the occurrence of an external interrupt from the clock or some I/O device. Then, the following state will be the result of the execution of the atomic action of that process immediately after the action that caused it to become blocked.

Our assertion language is an extended temporal logic including two new operators for describing properties involving real time. These operators are two of the more useful specializations of general real-time eventuality operators. Koymans, Vytopil and de Roever [Koymans 83] discuss a very general operator: $U_{=t}$, which they call "strong until in real-time t."

Our assertion language includes the familiar operators of temporal logics. Assertions may refer not only to the values of program variables but to program locations as well. The assertions that refer to this locations are "*at* A," "*in* A," and "*after* A," where A is the label any statement of the program in question. Informally, they state that in some process control resides at the beginning of, in the middle of, or immediately following the statement A.

General temporal assertions are built up from immediate assertions (assertions on individual states) using the temporal operators "$\Box$" meaning "maintains," "$\Diamond$" meaning "eventually," and "$\bigcirc$" meaning "next," as well as the logical operators "$\wedge$," "$\vee$" and "$\neg$." Formulae involving these operators are assigned meaning recursively in the usual way:

$P \Box Q$	as long as P remains true, Q will be true
$\Box P$	henceforth P is true ($\triangleq$ **true**$\Box P$)
$\Diamond P$	now or later P is or will be true
$\bigcirc P$	in the next state P will be true

For brevity and clarity we will make use of a number of abbreviations. The first is in common use, the others are not. The first captures the notion of temporal implication or eventuality.

$$P \leadsto Q \equiv \Box(P \supset \Diamond Q).$$

Thus, "$P \leadsto Q$" means that "whenever P is true, Q will eventually become true." The next abbreviation captures the notion of "next to execute", i.e. that a particular action will be the next selected for execution.

$$\mathit{sel}\ A \equiv \mathit{at}\ A \wedge \bigcirc\, \mathit{after}\ A,$$

where A is the label of an indivisible operation. Thus "*sel* A" is true if A is eligible for execution (*at* A) and will have been executed by the next state in the sequence ($\bigcirc$ *after* A). For a given action, we indicate that its execution must be followed by another execution of the same action by:

$$\mathit{sel}\ A \leadsto \bigcirc \mathit{sel}\ A,$$

which indicates that A must be selected for execution infinitely often. The $\bigcirc$ operator is required in this formulation by the structure of the logic.

The last abbreviation presented here is the predicate "*path*." The expression "*path*(P, Q, R)" denotes the property: "if execution reaches a state satisfying P, then if it is to reach a state satisfying R, it must first pass through a state satisfying Q." This is expressible using the non-abbreviated form of the "$\Box$" operator as:

$$P \supset \neg Q \vee R \,\Box\, \neg R.$$

In addition to the more familiar notation of temporal logic just discussed, we now review our notation [Bernstein 81b] for real-time properties. Our approach is to define "time bounded eventuality" properties similar to the "$P \leadsto Q$" defined above. We intend that these do no more than put simple upper or lower bounds on the time between P and Q. For assertions I_1 and I_2:

$I_1 \overset{<n}{\rightsquigarrow} I_2$	iff	Every occurrence of I_1 is followed by an occurrence of I_2 in fewer than n time-units.
$I_1 \overset{>n}{\rightsquigarrow} I_2$	iff	No occurrence of I_1 is followed by an occurrence of I_2 except in more than m time-units.

These two properties are useful for specifying upper and lower bounds for program execution. Note that the second does not imply liveness, i.e. it does not assert that a state satisfying I_2 will ever be reached. Thus, it is only an assertion of a lower bound.

We now review a few of our inference rules for reasoning with real-time to indicate the flavor of the reasoning involved. The first gives the result of a comparison of relative rates of progress.

$$\frac{A \overset{<n}{\rightsquigarrow} B,\ C \overset{>n}{\rightsquigarrow} D}{path(A \wedge C,\ B,\ D)} \qquad \text{RPR}$$

This asserts that if it takes less than n units time for B to become true starting in a state in which A is true and more than n units for D to become true from any state in which C is true, then if A and C are true simultaneously, then B must become true before D becomes true. This is a simple consequence of the relative speeds of achieving B and D starting from a state in which A and C are true.

To compute execution speeds, we must be able to combine the rates associated with parts of a program to give rates for their combination. This is done using rules such as the following:

$$\frac{P \overset{>n}{\rightsquigarrow} Q,\ Q \overset{>m}{\rightsquigarrow} R,\ path(P,\ Q,\ R)}{P \overset{>m+n}{\rightsquigarrow} R} \qquad \text{GTA}$$

This states that if it takes at least n units to get from P to Q and m from Q to R that it takes at least $m+n$ from P to R. The *path* assertion is necessary to disallow the case where execution could go through R on the way from P to Q. A similar rule (LTA) applies for the "$P \overset{<n}{\rightsquigarrow} Q$" relation except that no *path*

assertion is necessary in that case.

Suppose that $P \rightsquigarrow Q$ via either of two paths, and that we can derive the execution rates for each path, then the following rule gives the rate for the execution from P to Q.

$$\frac{(P \wedge \neg B) \overset{<n}{\rightsquigarrow} Q,\ (P \wedge B) \overset{<m}{\rightsquigarrow} Q}{P \overset{max(n,m)}{\rightsquigarrow} Q} \qquad \text{ALP}$$

This concludes our brief review of our notation and previous work. It is hoped that the reader now has a sufficient feel to enable him to understand the presentation to follow.

3. Level Structured Systems and Analysis

As mentioned in the introduction, we consider here real-time systems that have been designed with a level structuring and present a technique for deriving bounds on response times in these systems. While it is certainly possible to apply previous techniques to the analysis of such systems, ignoring their structure, the resulting task is likely to be far too complex to allow a reasonable chance for success. Thus, in this work, we consider systems that are divided into levels by priority, where each level is made up of one or more processes of a given priority, and is autonomous with respect to other levels. Levels have disjoint address spaces and may communicate only via procedure calls.

In systems involving several processes running at different priorities, the lower priority processes are interrupted and lose control of the processor whenever a higher priority process is ready. Thus, in these systems, code sequences in lower priority levels that would execute without interruption were it not for the higher priority levels (i.e. do not contain any statements that cause process switching) are subject to arbitrary interruption by higher levels. While the execution times of indivisible operations are not affected by the priority structure, the derivation of response times for sequences of indivisible operations at any but the highest priority (non-interruptible) level is made extremely difficult by the ever present potential for interruption.

Rather than consider the potential time lost (due to of pre-emption by higher levels) between each pair of indivisible operations of a particular code sequence, it is often possible to consider the effects of the higher priority levels on the lower levels as a net increase in the execution times over sequences of code. That is, one can derive the execution time for a particular code sequence in the absence of any higher priority processes, and then increase it by the amount that higher priority processes would use up during that time. In effect one is treating the higher levels as having a slowing influence over the entire sequence rather than accounting for their effects piecemeal.

The analysis of the level-structured systems considered here, parallels their structure. Each level is examined independent of the others in order of decreasing priority. Once a level has been analyized, the results for that level are adjusted to reflect the influence of higher levels. This will be covered in detail in the next two subsections.

3.1. Level Structured Systems

Each level contains processes that spend some time executing and the rest of the time waiting. Thus, process execution is cyclic in nature, being divided into an execution phase and a waiting phase. This is a fairly restrictive model of processes and simplifies the discussion and analysis presented here. Similar analysis could be carried out even if the cycles were more complex, perhaps involving several execution phases of varied length rather than just one. The length of the execution phase depends on the code to be executed and the activities of processes at higher levels. The time required for a complete cycle can depend on many factors as will be seen in detail later.

Processes wait either voluntarily, as a result of executing some operation that explicitly causes a process switch (eg. a **wait**) or involuntarily, because of pre-emption by processes at higher priority. The lowest and highest priority levels represent special cases. Processes at the lowest priority level need not wait voluntarily, as processes at all other levels must, since they interrupt no one. (If a process at some higher level never waited voluntarily, then processes at all lower levels would make zero progress. This yields very poor real-time bounds for these levels.) Processes on the highest priority level, on the other hand, only wait voluntarily since there are no higher priority levels to pre-empt them.

A level may also contain passive procedure code implementing functions that are made available to processes at the next lower level. Since the code resides at a higher level than the calling process, that process has its priority raised for the duration of the call. As a result, no other process at the callers level may execute during the execution of the procedure body. This can have the effect of making procedure calls to higher levels "atomic" with respect to the calling level. A discussion of the implications and benefits of this implementation is given by Bernstein and Ensor [Bernstein 81a].

The question of "atomicity" of procedure calls affects the analysis of level structured systems. If procedure code contains statements that lead to process switching, such as **pause** (delays a process for a specified period) or **wait** (delays a process pending a particular action, **send**, by another process), then there can arise situations where execution dies out on the called level and some other process in the calling level can be allowed to execute before the call actually completes execution. This by itself is not an obstacle to level-structured analysis. However, it is necessary to place some restrictions on this situation in order to apply our results. This is explained below.

Our level structured approach to analysis requires that response times in a level be derivable from that level and higher levels. Consider the case where a process executing in level k calls a procedure on level $k-1$ (higher priority levels have lower numbers) that contains a **wait**. If the corresponding **send** is executed as a result of a future call from level k to $k-1$ by some other process, then the execution time for the first procedure can not be derived by examination of levels 1 through $k-1$ and our techniques cannot succeed. Thus, for our current purposes, we assume that whenever a procedure contains a **wait**(q), the corresponding **send**(q) must be executed as a result of actions on the part of another process on the same (called) level and not on the calling level.

To illustrate this distinction, we consider two example cases. In the first case, there is a procedure which is called by a lower priority level process to obtain some service from a process on the higher priority level. If that process is currently busy, the calling process could execute a **wait** statement and be reactivated by the server process when it is no longer busy. In this case, the time that the calling process spends waiting depends on the server process and not on the callers level.

On the other hand, one could imagine a situation wherein the higher priority level procedure protects a shared resource used by the lower priority level processes. When a process desires to use the resource it calls the procedure to see if the resource is available. If not, the process executes a **wait** statement to delay itself until some other lower priority level process (the one currently in possession of the resource) calls a procedure to return it and wake up the waiting process. In this case, the time the first process spends waiting for the second to call and return the resource cannot be determined without examining the code at the lower priority level. Thus, this situation is not allowed.

3.2. Level by Level Analysis

We now present a technique for the analysis of the level structured systems just described. For an N level system, the analysis proceeds from the highest priority level (1^{st}) to the lowest (N^{th}). At each level, analysis is performed to derive results of two types. Some results are of immediate interest in that they yield information that may be desired about the level. Other results are required for the further analysis of lower priority levels.

The basic strategy in examining a particular level is to derive response times for that level, as if that level represented the entire system. This may be done, for example, using the techniques outlined briefly in section 2. These results are "virtual-time results," and are subject to adjustment to account for the time used by higher priority levels, before they can be considered valid response time results for the system as a whole or "real-time results."

As described earlier, processes are cyclic. Thus, the slowing effect of a higher priority level process on the timing characteristics of processes on a lower priority level is captured by the ratio of the execution time in each cycle to the total cycle length. Consider the two processes shown in Figure 1. The high priority process P_1 executes for 20 milliseconds out of every 100 milliseconds and the code sequence from A to B in P_2 would require 1 second if executed without interruption. With P_1 present, P_2 loses these 20 milliseconds out of every 100, so that it executes at $1-\frac{20}{100}$ of the rate at which it would execute alone. Thus, P_2 will require 1.25 seconds on the average to execute from A to B. The actual value will range from 1.24 to 1.26 depending upon where P_1 is in its cycle when P_2 begins at A. The techniques presented here will give one or the other of these depending upon whether one is deriving an upper bound or a lower bound, but uses the same idea and is based on these ratios.

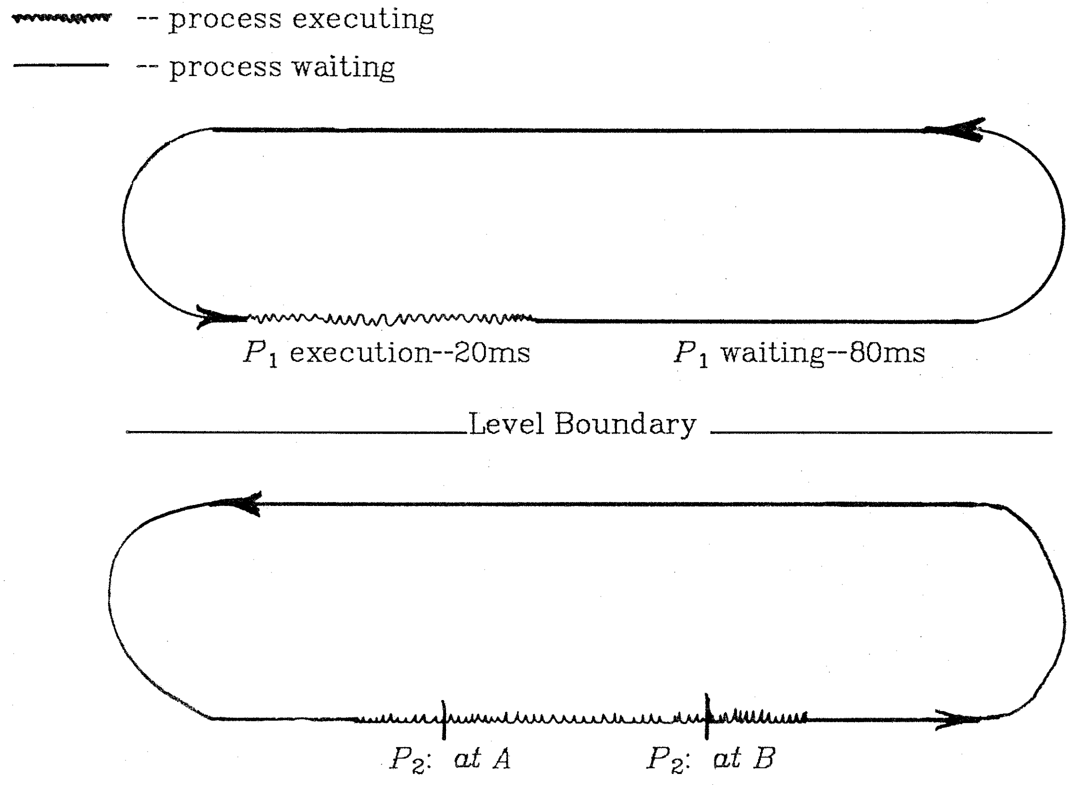

Figure 1 -- Process Time Dilation

In examining a level in isolation, it is not possible to derive response-time properties for sequences of code in that level that involve procedure calls to higher priority levels, since the execution times for these procedures depend on code in the higher priority level. For such sequences, one first derives virtual-time results for the code sub-sequences between the inter-level calls. These virtual-time results are then adjusted for the effects of higher priority levels to produce real-time results. These real-time results can then be combined with the real execution times for the called procedures previously obtained by analysis of the higher priority level. The reason for this two step approach is that the procedure code resides at a higher priority level and is thus less susceptible to interruption by higher priority levels than is the calling code. This will be discussed in detail as we discuss the derivation of load-pairs for a level.

The techniques for deriving properties of the form "$A \overset{<n}{\rightsquigarrow} B$," called "upper bounds," are very nearly the dual of those for properties of the form "$A \overset{>m}{\rightsquigarrow} B$," called "lower bounds," though not precisely. We formulate the following discussion in terms of proving an upper bound property for a level-structured system, though we will necessarily cover the material necessary for lower bound properties along the way.

Assume that we have an N-level system and wish to prove that an upper bound result $REQUESTED \overset{<T}{\rightsquigarrow} COMPLETED$ holds for the system, where *REQUESTED* and *COMPLETED* are predicates describing the states of processes in level k $(1<k<N)$. Further, assume that the execution path taking the system from the former state to the latter is an arbitrary execution of processes in levels k through 1. We need not consider execution in levels $k+1$ through N, since they contain lower priority process whose execution cannot affect timings at level k. To derive this property we must apply the following level-by-level analysis to levels from 1 to k, since lower levels do not influence the result.

For each level, we must derive both real-time and virtual-time results that abstract the response time characteristics of that level for use in deriving results for lower priority levels.

Note--In order to keep our notation clear, we will adopt the convention that virtual time processes are indicated using a slightly different operator. We will write:

$$P \overset{<n}{\rightsquigarrow\!\!>} Q,$$

to denote the virtual-time result that P leads to Q. This result may be converted to a real-time result as described below, which will then be written using the familiar notation and a different value for n.

Processes execute a "wait/execute-cycle", which we abstract as two "load-pairs." For each process P_i, a ***max-load-pair*** is a pair $<c_i, E_i>$, where c_i is a lower bound on the duration of the cycle and E_i is an upper bound on the actual execution time during each cycle. The max-load-pair for P_i represents the maximum load to the system that can be caused by P_i. Analogously, ***min-load-pair*** $<C_i, e_i>$ represents the minimum load possible due to a process, where now C_i is an upper bound on the cycles duration and e_i is the minimum actual execution time during the cycle. The load-pairs are combined into min and max load-sets for each level:

$LSmin_k \triangleq \{<C_i, e_i> \mid P_i \textit{ is at priority higher than } k\}$ and

$LSmax_k \triangleq \{<c_i, E_i> \mid P_i \textit{ is at priority higher than } k\}$,

which represent the total system load due to levels higher than k.

Given a virtual-time property for some level, we wish to adjust it to take into account the time used by higher priority levels. A load-set is a way to represent the percentage of time used up by higher levels and hence is an indication of the degree to which those levels ''slow down'' processes at the current level. The load-sets for a given priority level are used to adjust virtual-time results for that level. Virtual lower bounds are adjusted using the min-load-sets since, in the worst case, the higher priority level processes will all execute as short a time as possible and as seldom as possible. For virtual upper bounds we use max-load-sets since the adjusted real upper bound must hold even when all the higher priority level processes are executing as long as possible and as often as possible.

> Note--For lower bound properties ($A \overset{>m}{\rightsquigarrow\!\!\gg} B$), of course, no adjustment is required since any virtual lower bound must also hold as a real lower bound if the level must ''run more slowly,'' however, adjustment will result in tighter lower bounds which are more useful. End-of-note

The adjustment of a virtual-time property results in a real-time property, which is the same property with a corrected (real) bound ($<n'$ or $>m'$). In order to assure the validity of the adjusted result, upper bounds must reflect the maximum possible dilation, while lower bounds receive the minimum required dilation. Thus, no attempt is made to reflect average case behavior; worst case (or worse) is generated.

The algorithm to follow generates the new upper ($<n'$) or lower ($>m'$) bound for virtual upper (n) or lower (m) bounds at level k as defined by the relations:

$$n' \triangleq \min\{d \mid d = n + \sum_{<c_i,\, E_i> \in LSmax_k} \left\lceil \frac{d}{c_i} \right\rceil * E_i\}, \qquad \text{and}$$

$$m' \triangleq \min\{d \mid d = m + \sum_{<C_i,\, e_i> \in LSmin_k} (\left\lfloor \frac{d}{C_i} \right\rfloor + \text{bump}(d, C_i, e_i)) * e_i\},$$

where: $\text{bump}(x, y, z) \triangleq \begin{cases} 1 & \textit{if} \;\; x \bmod y > y - z \\ 0 & \textit{otherwise} \end{cases}$

Intuitively, these just say that the adjusted upper (lower) bound is the lowest number d that represents a time sufficient for the uninterrupted execution (n or m) as well as the total of the maximum (minimum) execution times for the maximum (minimum) number of cycles that can possibly (must necessarily) occur during that time. The maximum number of cycles of process P_i for adjusting upper bounds is given by "$\left\lceil \frac{d}{c_i} \right\rceil$."

For lower bounds, the minimum number of complete cycles (and hence execution phases) of a process P_i during a given period will be realized when that period begins immediately upon completion of one of P_i's execution phases. Then, after $\left\lfloor \frac{d}{C_i} \right\rfloor$ cycles, P_i can take as long as $C_i - e_i$ to begin another execution phase, since that is the latest point at which the execution phase can begin and still complete by the end of the cycle at C_i. Thus, the minimum number of execution phases of process P_i for adjusting lower bounds is given by "$\left\lfloor \frac{d}{C_i} \right\rfloor + \text{bump}(d, C_i, e_i)$." where the function "bump" accounts for the assumption that the execution phase begins as late as possible.

The algorithm for computing the adjusted upper (lower) bounds works by successive approximation to find the values defined above. For each candidate n' (m'), a test is made to see whether n (m) and the executions of the maximum (minimum) number of cycles that can occur during n' (m') can fit in n' (m'). If not, then n' (m') is set to the smallest value such that that number of cycles of execution will fit. Changing n' (m') however, lengthens the time during which cycles could occur. As a result, the new value of n' (m') may not satisfy the above test. Thus we must iterate until an n' (m') is reached that satisfies the above test. If, for the current level, it is the case that

$$\sum_{<c_i,\, E_i>\in LSmax_k} \frac{E_i}{c_i} \quad \text{or, for lower bounds:} \quad \sum_{<C_i,\, e_i>\in LSmin_k} \frac{e_i}{C_i},$$

which represents the fraction of the CPU left for processes at level k, is less than one, the value of n' (m') will increase faster than the execution time required by the additional cycles, and the algorithm will converge. The algorithm is shown in Figure 2.

Under certain circumstances, the algorithm converges more quickly by starting with an approximation for n'. For computing adjusted upper bounds, the relation:

$$n' \approx n * \left(\frac{1}{1 - \sum_{<c_i,\, E_i>\in LSmax_k} \frac{E_i}{c_i}} \right),$$

provides an approximate n'. The accuracy of this approximation increases as:

(1) The ratio $\frac{E_i}{c_i}$ becomes smaller for all i, and

(2) The ratio $\frac{n}{c_i}$ becomes larger for all i.

That is, as there is less and less execution for each cycle, and as there are more and more cycles occurring during n'. In his paper [Wirth 77a], Wirth assumed these ratios to be very favorable, on the order of 10:1. This assumption allows the above relation to be considered an approximate solution to the problem of computing time dilation as was done by Wirth. The approximation may be viewed as an average case result. As we are interested in true upper bounds, this approximation will be too small and may thus be used as a starting point for our algorithm. For lower bounds, on the other hand, this relation tends to exaggerate the minimum possible dilation, and hence will lead to inaccurate results.

Assuming that the code for all higher priority levels has been analyzed, previously, so that the required information is available, we now describe how to generate the corresponding information from the current level in order to continue the level-by-level analysis.

(1) Compute:

$$\sum_{<c_i,\, E_i> \in LSmax_k} \frac{E_i}{c_i} \quad \text{or, for lower bounds:} \quad \sum_{<C_i,\, e_i> \in LSmin_k} \frac{e_i}{C_i}.$$

If it is greater than one, then the load from the higher priority levels is too high and the virtual bound becomes an infinite real bound. Quit now, the algorithm will never terminate if you proceed.

(2) Set n' (m') equal to n (m).

(3) Assign:

$$n' := n + \sum_{<c_i,\, E_i> \in LSmax_k} \left\lceil \frac{n'}{c_i} \right\rceil * E_i$$

or, for lower bounds:

$$m' := m + \sum_{<C_i,\, e_i> \in LSmin_k} \left(\left\lfloor \frac{m'}{C_i} \right\rfloor + \text{bump}(m',\, C_i,\, e_i)\right) * e_i$$

Now, if

$$n' < n + \sum_{<c_i,\, E_i> \in LSmax_k} \left\lceil \frac{n'}{c_i} \right\rceil * E_i$$

or, for lower bounds:

$$m' < m + \sum_{<C_i,\, e_i> \in LSmin_k} \left(\left\lfloor \frac{m'}{C_i} \right\rfloor + \text{bump}(m',\, C_i,\, e_i)\right) * e_i$$

then go back to (3) and do it again. Otherwise, you are done and the current value of n' (m') is the adjusted real-time bound.

Figure 2 -- Algorithm for Time Dilation

Recall that a load-pair has two components, the cycle time and the execution time per cycle. For both min-load-pairs and max-load-pairs the execution time must be a virtual-time result, while the cycle time must be a real-time result. The motivation for this derives from the algorithm for time dilation presented in the last subsection as we now explain.

The execution time represents the number of processor cycles lost to lower priority level processes during each cycle of the higher priority process. If the execution time component were required to be a real-time result, then inaccuracies would arise in the computation. Consider the dilation at level k due to a higher priority level l ($l<k$). The real-time bound on the execution during any cycle of a process in level l comprises the time that that process actually spends executing as well as the time that it loses to higher priority levels. In computing dilation for level k, we use $LSmin_k$ or $LSmax_k$, which contain entries for all levels having priority greater than k, including l. If these entries contained real-time execution bounds, then the effect on level k would "count the higher priority levels more than once" by dilating by higher priority levels as well as by times from level l that have been dilated by those same higher priority levels.

The cycle time, on the other hand, must be a real-time quantity, since it is used to compute the number of execution cycles that occur during a real-time interval. Thus, in deriving the load-pairs for a level, we must derive both real-time and virtual-time bounds.

Further, for general analysis, we will need both min-load-pairs and max-load-pairs, which requires that we have both lower and upper bounds on both the execution time and the cycle time for each process. Thus, to simplify notation, we introduce the abbreviation:

$$P \underset{lb}{\overset{ub}{\leadsto}} Q \equiv P \overset{<ub}{\leadsto} Q \wedge P \overset{>lb}{\leadsto} Q.$$

We define $startx_i$ and $donex_i$ to be predicates indicating the beginning and ending of the execution phase of process P_i. Then, for the min-load-pairs and max-load pairs for P_i we need to derive the virtual-time result:

$$startx_i \underset{e_i}{\overset{E_i}{\leadsto\!\!>}} donex_i, \qquad \text{(execution-time)}$$

and the real-time result:

$$startx_i \underset{c_i}{\overset{C_i}{\leadsto}} \bigcirc startx_i. \qquad \text{(cycle-time)}$$

Then, for process P_i, the min-load-pair is $<C_i, e_i>$ and the max-load-pair is

$<c_i, E_i>$.

We also define, for each procedure entry F_j, the predicates *enproc$_j$* and *exproc$_j$* denoting the entry to and exit from procedure F_j resulting from a call. Then, for each procedure entry on level k, we will derive:

$$enproc_j \underset{t_j}{\overset{T_j}{\rightsquigarrow}} exproc_j, \quad \text{and} \qquad \text{(real-procedure-time)}$$

$$enproc_j \underset{t_j}{\overset{T_j}{\rightsquigarrow\!\!\gg}} exproc_j. \qquad \text{(virtual-procedure-time)}$$

This result will be used in the derivation of bounds for code sequences at level $k+1$, which involve inter-level calls to level k. As will be seen later, we will need this information for procedure entries in both real-time and virtual-time form.

The derivations of execution-time and procedure-time are similar, and will be discussed first, whereas cycle-times involve special consideration and will be dealt with separately. For any segment of code in level k, virtual upper or lower bounds are calculated as described above. Code sequences involving inter-level procedure calls pose no extra difficulties for virtual time results. As we assume that analysis has already been carried out for higher priority levels, the virtual times T_j and t_j are available for any called procedure and can be used as the bounds for the call statement. In this way, procedure calls are treated the same as any other statements with the exception that their virtual execution bounds are derived from the higher level rather than being determined by the language implementation.

> Note--As far as level k is concerned, these calls may or may not actually appear as atomic. What is important is the assumption, stated earlier, that the procedure-time for entries in level $k-1$ are completely determined by levels 1 through $k-1$ and do not depend on levels k through N. As long as this holds, the virtual execution time derived from the analysis of the higher priority levels can be used as described to derive bounds for code containing inter-level calls. If a procedure contains statements such as **wait** or **pause**, which can lead to execution in level k before the call terminates, then that procedure call must be treated as a non-atomic statement with a non-deterministic switching point in the

middle. Thus, in examining level k, this call must be seen as terminating an "atomic action" of its process. End-of-note.

Unfortunately, we can not take a virtual bound derived as above, and apply the algorithm of the last sub-section to generate a real bound, since the dilation for the procedure code above level k is different from that of the code in level k. The reason for this is that there are fewer levels that can pre-empt the execution of the procedure code. For this reason (see Figure 3), we must break any segment of code containing procedure calls into smaller segments bounded by procedure calls. The virtual bounds for these sub-segments ($sel\,1 \underset{a}{\overset{A}{\leadsto}} sel\,2$, $sel\,3 \underset{b}{\overset{B}{\leadsto}} sel\,5$) are then dilated using the above algorithm to yield the real-time bounds (a', A' and b', B'). These real-time bounds can then be combined (using LTA and GTA) with the real bounds for the procedures (t_1, T_1 and t_2, T_2), which were obtained by previous analysis to yield a real-time result:

$$sel\,1 \underset{a'+b'+t_1+t_2}{\overset{A'+B'+T_1+T_2}{\leadsto}} sel\,6.$$

Note that it does not matter here whether the procedures are actually executing or possibly waiting during the times given. For the purpose of finding bounds for calling processes, the only important factor is the time required before the procedure returns.

Thus we see that by combining techniques for dealing with levels in isolation with the algorithm above, we can derive procedure-times and execution-

```
1)    ... statements ...
2)    call proc1;        {real: $enproc_1 \underset{t_1}{\overset{T_1}{\leadsto}} exproc_1$ }
3)    statement
         ...
4)    statement
5)    call proc2;        {real: $enproc_2 \underset{t_2}{\overset{T_2}{\leadsto}} exproc_2$ }
6)    ... statements ...
```

Figure 3 -- Code sequence containing interlevel calls

times for the processes and procedures of level k.

For cycle-times, there are two very different cases. The execution for any cycle may be triggered either spontaneously, or in response to some action of the process. In the former case, the rate of occurrence of interrupts is independent of the execution rate of the level. An example of this might be a process that is actuated in response to a line-clock interrupt. This process will spend some time in execution and then simply terminate, only to be reactivated (modeled as the spontaneous transfer of the first label of the process from the delay-set into the ready-set) when the interrupt occurs. These processes can also be thought of as procedures (i.e. passive code), which are invoked spontaneously from outside the system since they simply terminate and don't execute a **wait** statement. There is no difference in the analysis, only the aesthetics. For these processes, the cycle-times (C_i and c_i) cannot be derived but are part of the environment and must be taken as given. Generally, the specification will be in terms of some range, in which case C_i is just the upper bound on the repetition time and c_i is the lower bound.

The second case is where the execution phase of a process occurs as a result of some triggering action taken by that process during its last execution phase. Examples of this are where a process initiates an I/O operation and then **waits** on the device signal, or where a process executes for a time and then executes a **pause** statement to delay itself for some interval. In these cases, the cycle time is not fixed by the environment but is dependent both on the length of time the process spends waiting and on the time it takes for the process to get around to taking its initiating action. For example, since we assume that the completion of an I/O operation does not result in the immediate transfer of control to the process waiting for it, a process that alternately **reads** and calculates can have either of two very different life cycles. If we take the process P from the time it begins its **read**, then if the system is very quiet, the cycle may be:

1) P waiting for **read** to complete
2) P **read** completed; P selected for execution
3) P processing new information
4) P finished processing
5) P begin **read** statement

On the opposite extreme, consider the case where the process P begins its **read**, and then, just before the **read** completes, a process on the next lower level ($k+1$) executes a procedure call to level k. At the same time, all the other processes on level k (who were waiting, since otherwise level $k+1$ could not have executed the call) enter the ready-set. Again, from the same point:

1) P waiting for **read** to complete, process on level $k+1$ executing
2) process on level $k+1$ calls procedure on level k
3) P's **read** completed
4) procedure at level k executing; all other processes become ready
5) procedure finishes executing; other process selected to run
6) all other processes run
7) last other process completes; P selected for execution
8) P processing new information
9) P finished processing
10) P begins next **read** statement

Of course, this is not actually the worst imaginable case, since one can imagine that P might never get another chance to run. In such a system it would be impossible to derive any upper bound. However, for a realistic situation, where processes are scheduled such that no process can have two chances to run before a given process has a chance, i.e. there is no overtaking, one must still consider this as a possible case.

Thus, for the derivation of cycle times for processes with dependent cycles, the upper and lower bounds are very different. For the lower bound (c_i), we assume the first scenario above and use:

$$c_i = e_i + IOTmin_D,$$

where $IOTmin_D$ is the minimum possible response time for the I/O device D as described in section IV. For the upper bound, we want to use the longest possible cycle and consider the second scenario. Recalling that T_j is the execution time of procedure entry F_j and assuming the "fair" scheduler (no overtaking) described above, we use:

$$C_i = E_i + IOTmax_D + \sum_{P_{l \neq i} \in \, level\ k} E_l \; + \max\{T_j \mid \text{for } F_j \text{ } an\ entry\ on\ level\ k\},$$

where the second two terms represent the maximum delay in being selected for execution. Wirth [Wirth 77a] called the third term a "hidden delay" in responding to I/O interrupts; he did not consider procedure calls.

This concludes the description of the analysis. The next section presents an example showing the application of these techniques to a simple level-structured system.

4. Example

In this section, we illustrate the techniques presented in the previous section for the analysis of level-structured real-time systems. We examine a system for process monitoring in which there are two levels as seen in Figure 4. Note the introduction of the structuring commands for dividing the system into levels (**level ... levend**) and the keyword **function** for defining functions. The former cause all bracketed code to run at the same priority level (lower numbers have higher priorities), while the latter has its familiar meaning.

```
level 1
function GETIME(): current-time;
   ... { read day/date record, calculate current-time and return} ...
   end;
cobegin
   begin                                   {process CLOCK}
      ... {update day/date record by one "tick" } ...;
      end;
// while true do begin                     {process FLOW_MON}
      pause($w_{FLOW\_MON}$)
      ... {read flow sensor value} ...;
      ... {process new value} ...;
      end
coend
levend;

level 2
   while true do begin                     {process TEMP_MON}
      pause($W_{TEMP\_MON}$);
      time := GETIME();
      ... {read temperature sensor value} ...;
      ... {process based on value} ...
      end
levend
```

Figure 4 -- A level-structured system for process monitoring.

The higher priority level, level 1, contains two processes CLOCK and FLOW_MON as well as a function for reading the current time called GETIME. CLOCK is a process that receives an interrupt from an external timer at regular intervals, at which time it updates the current value of the *day/date* record. FLOW_MON is a process which reads the current value of a flow sensor from its memory location and does some processing based on the relation of the current value to the previously found value. After this processing, FLOW_MON puts itself to sleep using a **pause** statement for a predetermined interval. The function GETIME is supplied so that processes at the lower level can read the *day/date* record without being interrupted and (possibly) getting inconsistent results. Note that to avoid this possible inconsistency, GETIME cannot contain **wait** statements.

Level 2 runs at lower priority and contains one process TEMP_MON, which alternately waits (**pause**) and does processing. Each time TEMP_MON wakes up, he checks the time by calling GETIME and reads the current value of the temperature sensor (this sensor is sampled at lower priority since it monitors a more slowly changing quantity) and does some processing based on its value.

We wish to show for this example that the cycle time for the TEMP_MON process is short enough to ensure that it will be able to read the temperature sensor at a rate high enough so as not to lose information. The possibility exists that this process be pre-empted so much by the higher priority CLOCK and FLOW_MON that its processing time is dilated to the point where, after **pause**ing, it has missed a change in the sensor value. We wish to show that this is not the case.

We assume in this example that execution times for simple statements are significant and that analysis has already been carried out on the code sequences indicated in Figure 4 by the ellipses and comments. The presumed results are as shown in the accompanying table. Note that the results for the **pause** statements are real-time results based on the semantics of the **pause** statement. The real-time result for CLOCK is assumed to have been given for the environment and gives the rate at which CLOCK-interrupts occur.

We are interested in deriving the rate at which TEMP_MON can read the temperature in order to show it is high enough to ensure that no information is lost. In order to do this we must derive the real-time bound:

Module	Real/Virtual	Presumed Results
GETIME	virtual	$enproc_{GETIME} \overset{<T_{GETIME}}{\rightsquigarrow\!\!>} exproc_{GETIME}$
CLOCK	real	$sel\ 6 \overset{>c_{CLOCK}}{\rightsquigarrow} \mathrm{O}sel\ 6$
	virtual	$sel\ 6 \overset{<E_{CLOCK}}{\rightsquigarrow\!\!>} after\ 8$
FLOW_MON	real	$sel\ 10 \underset{w_{FLOW\_MON}}{\overset{W_{FLOW\_MON}}{\rightsquigarrow}} after\ 10$
	virtual	$sel\ 11 \underset{e_{FLOW\_MON}}{\overset{E_{FLOW\_MON}}{\rightsquigarrow\!\!>}} sel\ 10$
TEMP_MON	real	$sel\ 18 \overset{<W_{TEMP\_MON}}{\rightsquigarrow} after\ 18$
	virtual	$sel\ 20 \overset{<X_{TEMP\_MON}}{\rightsquigarrow\!\!>} sel\ 18$

$$sel\ 20 \overset{<C_{TEMP\_MON}}{\rightsquigarrow} \mathrm{O}sel\ 20,$$

and compare C_{TEMP_MON} to the desired time between reads.

We first finish the analysis of level 1 and then handle level 2. Since level 1 is the highest level, it loses no time due to pre-emption by higher levels. Thus, the virtual-time results for GETIME, CLOCK and FLOW_MON are also real-time results. Since we are deriving an upper bound for TEMP_MON in level 2, we need to construct the max-load-pairs for CLOCK and FLOW_MON to find the load-set $LSmax_2$. For CLOCK, we have the lower bound on the cycle-time given from the environment and the execution time from prior analysis giving: $<c_{CLOCK}, E_{CLOCK}>$. For FLOW_MON, the (real) minimum cycle time is given by: $c_{FLOW\_MON}:=w_{FLOW\_MON}+e_{FLOW\_MON}$, while the (virtual) execution time we have from prior analysis. Thus, we have $<c_{FLOW\_MON}, E_{FLOW\_MON}>$.

For TEMP_MON, the (real) upper bound on the cycle time is derived using LTA from the (real) upper bounds around his loop. The (real) upper bound for the **pause** is as shown in the table, and the (real) upper bound for the procedure call on line 19 is also shown, as discussed above. The real upper bound for the code in lines 20 and 21 is calculated using the time-dilation algorithm presented above, where

$$LSmax_2=\{<c_{FLOW\_MON}, E_{FLOW\_MON}>, <c_{CLOCK}, E_{CLOCK}>\}$$

and the starting virtual bound is $<X_{TEMP\_MON}$ as shown in the table of assumed results. This results in the real-time result X'_{TEMP_MON}. Having derived all the necessary results, we can use LTA to derive:

$$sel\ 20 \overset{<C_{TEMP\_MON}}{\rightsquigarrow} \Diamond sel\ 20,$$

where now $C_{TEMP\_MON}=W_{TEMP\_MON}+T_{GETIME}+X'_{TEMP\_MON}$. To make the example concrete, suppose that the various time-bounds given in the table above have the following values.

Bound	Value	Bound	Value
E_{CLOCK}	4ms	T_{GETIME}	5ms
c_{CLOCK}	20ms	W_{TEMP_MON}	50ms
w_{FLOW_MON}	35ms	X_{TEMP_MON}	100ms
E_{FLOW_MON}	8ms		
e_{FLOW_MON}	5ms		

Then

$$LSmax_2=\{<20ms, 4ms>, <40ms, 8ms>\},$$

and the algorithm yields the result $X'_{TEMP\_MON}=176ms$. Then, the upper bound on the time for a complete cycle of the TEMP_MON process is 231ms. Thus, if the desired inter sample time on TEMP_MON's temperature sensor is 231 ms or greater, the system will meet its constraint.

5. Conclusion

We have presented a method based on an extended temporal logic for deriving response-time properties in complex systems with a level structure based on priority. The method involves a level by level examination of the system, where information distilled from each successive level is used to adjust the results for later levels. The results obtained at each level of the system are static, in that they are not affected by later analyses, which obviates having to consider a complex system as a whole.

The method is quite general and deals successfully with procedure calls between levels as well as systems whose execution characteristics make them inaccessible to simpler analysis. It is possible to derive both lower and upper bounds that are tight, in the sense that they are, in principle, achievable by the system in execution.

These analysis techniques have not yet been applied to actual, working systems, though we feel this is possible, perhaps with some mechanical aids. This remains a problem for the future.

References

[Bernstein 81a] A. J. Bernstein, J. R. Ensor.
A Modula Based Language Supporting Hierarchical Development and Verification.
Software -- Practice and Experience 11 (3):237-256, March 1981.

[Bernstein 81b] A. J. Bernstein, P. K. Harter.
Proving Real Time Properties of Programs With Temporal Logic.
Proceedings of the 8th Annual ACM Symposium on Operating Systems Principles (Asilomar, California), pages 1-11, ACM SIGOPS, December 1981.

[Gabbay 80] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi. On the Temporal Analysis of Fairness.
Conference Record of the 7th Annual ACM Symposium on Principles Of Programming Languages (Las Vegas), pages 163-173, ACM SIGACT-SIGPLAN, January 1980.

[Koymans 83] R. Koymans, J. Bytopil, W. P. de Roever.
Real-Time Programming and Asynchronous Message Passing.
Proceedings of the Second Annual Symposium on Principles of Distributed Computing, (Montreal, Quebec), pages 187-197, August 1983.

[Owicki 82] S. S. Owicki, L. Lamport.
Proving Liveness Properties of Concurrent Programs.
ACM Transactions on Programming Languages and Systems 4 (3):455-495, July 1982.

[Wirth 77a] N. Wirth.
Towards a Discipline of Real-Time Programming.
Communications of the ACM 20 (8):577-583, August 1977.

[Wolper 81] P. Wolper.
Temporal Logic Can Be More Expressive.
Proceedings of the 22nd Annual Symposium on Foundations Of Computer Science, pages 340-348, IEEE Computer Society, 1981.