Reasoning About Time in Higher-Level Language Software

ALAN C. SHAW

Abstract—**A methodology for specifying and proving assertions about time in higher-level language programs is described. The approach develops three ideas: the distinction between, and treatment of, both real-time and computer times; the use of upper and lower bounds on the execution times of program elements; and a simple extension of Hoare logic to include the effects of the passage of real-time. Schemas and examples of timing bounds and assertions are presented for a variety of different statement types and programs, such as conventional sequential programs including loops, time-related statements such as delay, concurrent programs with synchronization, and software in the presence of interrupts. Examples of assertions that are proven include deadlines, timing invariants for periodic processes, and the specification of time-based events such as those needed for the recognition of single and double clicks from a mouse button.**

Index Terms—**Hoare logic, interval analysis, real-time, timing analysis, verification.**

I. Introduction

REAL-TIME systems and many other computer applications must meet specifications and perform tasks that satisfy timing as well as logical criteria for correctness. Examples of timing properties and constraints include deadlines, the periodic execution of processes, and external event recognition based on time of occurrence (e.g., [9], [18]).

We present a scheme for reasoning with and about time and for specifying timing properties in concurrent programs. The objectives are to predict the timing behavior of higher-level language programs and to prove that they meet their timing constraints, through the direct analysis of program statements. Timing is *deterministic*, not stochastic, so that our results are applicable to "hard" real-time systems.

There is a clear distinction—one that is not always recognized—between timing *predictability* on the one hand and speed or efficiency on the other. Both are important, but our work is concerned primarily with predictability. Higher-level languages are being considered and used more frequently for constructing concurrent real-time software, for all of the standard reasons—they offer facilities for structuring data and control and for modularization, programs are easier to design and code, the results are easier to understand and maintain, and software is more portable. What is missing is the ability to predict the timing behavior of these programs and methods to reason about time within programs.

Two major ideas are developed. The first is that upper and lower bounds on execution times for statements can be derived, based on given bounds for primitive statements and elements in the language and underlying system. Schemas for obtaining bounds are presented for conventional sequential statements including loops, for timing-related statements that refer to imperfect computer clocks, and for synchronization and communications operations with timeouts. The second idea is to extend Hoare logic to include the effects of updating real-time (as defined by an ideal global clock) after each statement execution. Many examples are used to illustrate the techniques. The main contributions are the development and synthesis of these two ideas, and the demonstrations of program assertions and invariants involving time.

Several works are related to or have influenced our research. A methodology that has some similarities with ours was proposed by Haase [10]. His methods assume a concurrent programming language based on guarded commands, running on a non-von Neumann architecture; a deterministic execution time is given for each simple statement but execution times for conditional elements and iterations need not be defined. Dijkstra's weakest preconditions are employed for making and proving assertions. In our work, we assume a range of *conventional* machine architectures and develop time *bounds* to reflect execution of *control* (conditional, iteration), as well as data transforming operations. Also, unlike Haase, we distinguish between computer time and real-time, and analyze the timing behavior of timing-related statements and communications operations. Haase introduces real-time as a (fictitious) variable in his program space, that is updated to reflect exact execution times; we use the same idea initially and then derive running bounds for real-time. We have also been influenced by Jahanian and Mok who argue that the notion of a real-time or clock variable cannot be realized correctly because of concurrency problems [13], and by a presentation given by Pnueli who argues that clock variables are indeed feasible [20].

Manuscript received August 31, 1987; revised November 25, 1987. The principal part of this work was performed at Laboratoire MASI, University of Paris 6, during the academic year 1986–1987. Further work was done during a month's visit in Spring 1987 at the Institut fur Informatik, ETH, Zurich, and during Summer 1987 at the University of Washington. This work was supported in part by the above institutions, by a Fulbright research grant from the Franco-American Commission, and by the Washington Technology Center.

The author is with the Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195.

IEEE Log Number 8928290.

There does not seem to be any literature on how to derive execution time for statements in higher-level programming languages. (At the machine language level, one can use Knuth's techniques for conventional sequential programs [15].) An interesting attempt at designing a language with deterministic timing predictability is the ESTEREL project [4]; in ESTEREL, instructions take zero time except for those that explicitly deal with time and external events. Other than ESTEREL, it appears that predictability has not received much attention. In fact, part of the motivation for our research has been the lack of predictability in languages and systems proposed or used for real-time applications, Ada, of course, being the most outstanding example [8].

The next section presents the basis and elements of our techniques for dealing with time. Sections III, IV, and V then show how timing bounds and assertions can be produced for a variety of different statement types and programs. In Section VI, some additional practical issues related to interrupt handling and processor sharing are introduced. The concluding section discusses some undeveloped areas and next steps.

II. A Simple Logic for Handling Time

A. Real-Time and Computer Time

Our reasoning about time refers ultimately to an idealization of real-time as realized by a perfect global clock. This real-time is denoted by rt; it may be, for example, Greenwich Mean Time.

Computer time is the discrete approximation to real-time implemented on machines by a variety of hardware and software methods. At the hardware level, there may be fixed interval or programmable interval timers that produce "tick" interrupts, or absolute timers that periodically update a software-accessible counter [24]. A software clock would typically use the tick interrupts or the value of an absolute time counter to generate a computer time.

Many versions of computer time can coexist in a system. We will assume that each version is approximately synchronized with perfect real-time as follows. If ct represents a computer time, then

$$ct = rt + \delta$$

where $|\delta| \leq \epsilon$ and ϵ is determined by the accuracy of the hardware clock, tick interval, synchronization interval, and synchronization method (e.g, [17]). (This relation does not include the access time to obtain or compute ct.)

It has been assumed implicitly that our abstract real-time rt is represented by a real number and that each ct is a computer approximation to a real number. ct is normally a more complex data structure, with separate components designating, for example, the year, month, day, hour, minutes, and seconds. Updating a clock, computing with time as a variable, or even reading a computer clock can therefore consume a significant amount of time. These effects are considered in Section IV.

B. Initial Architectural Assumptions

A broad spectrum of hardware architectures is possible. There may be many processors with private or shared memory; these could communicate through a shared memory, directly over a bus, through a general communications network, or some combination of these. Programmed processes or tasks are considered static, with no dynamic creation at run-time except perhaps during an initialization period.

We assume that each software process has its own dedicated processors—i.e., no time-sharing of machines. This includes processes responsible for input and output. Similarly, each clock computes its version of time on a dedicated processor. While we will relax these assumptions later, they permit a simple basis for the treatment of time. Also, the one-to-one association between processes and processors is a feasible allocation scheme for systems where performance and timing predictability are critical, especially in an era of reduced hardware costs (e.g., [11]).

Thus, software interference from an operating system for such functions as dispatching, interrupt handling, and input–output will not occur. The effects of hardware sharing, for example, due to common memory or to bus contention, are factored into the execution times of program statements.

C. Execution Times for Statements

The aim is to analyze directly higher-level language programs. We assume a modern Algol-like language, augmented by functions for handling computer time, process synchronization and communication, input–output with the real-time external world, and standard input–output. A program consists textually of one or more statements, plus some declarative text.

A particular execution of a statement S will be delimited by two events: a *start* event and an *end* event. (Events are just points in time and consume no time.) Let $t(S)$ be the real-time between these two events. Ideally, one would like to know the *execution time* $t(S)$ for every execution of every statement S in a given program. Unfortunately, there is no way to determine $t(S)$ *a priori* in general. The value depends on the context of S, the data of the program, the compiler, the run-time system, the target machine, and possibly other things.

However, it is possible to obtain *bounds* for $t(S)$. Let

$$T(S) = [t_{\min}(S), t_{\max}(S)]$$

where $t_{\min}(S) \leq t(S) \leq t_{\max}(S)$ for *all* executions of S in a given program. In the worst case, $T(S) = [0, \infty]$; but one can almost always do better than this. Much of this paper is concerned with methods for finding tight bounds in $T(S)$ for various types of statements S. It should be emphasized that, like $t(S)$, $T(S)$ is also in general dependent upon the particular language, context, compiler, run-time system, and target machine.

Generally, $T(S)$ will be provided for elementary statements, expressions, and control structures S, and T is then

derived or computed for more elaborate constructs. A simple example is the sequential composition of two statements $S = S_1; S_2$, yielding the bounds

$$T(S) = T(S_1) + T(S_2)$$

where we define $[a, b] + [c, d] = [a + c, b + d]$. This assumes that the end event of S_1 occurs simultaneously with the start event of S_2. It is also conceivable that sequential control might consume time in a particular realization of a programming language. Including sequencing overhead, we obtain

$$T(S) = T(S_1) + T(S_2) + T(;)$$

where $T(;)$ is the time for sequencing S_1 and S_2. We will use the first interpretation throughout the paper (i.e., $T(;) = [0, 0]$).

Similarly, consider a system with two concurrent processes and associated programs S_1 and S_2, respectively. Denote $S = S_1//S_2$, where // indicates concurrent control. The execution time bounds are typically given by

$$T(S) = \max(T(S_1), T(S_2)) + T(//)$$

where $\max([a, b], [c, d])$ is defined as $[\max(a, c), \max(b, d)]$ and $T(//)$ is the overhead corresponding to running S_1 and S_2 in parallel.

At a lower level, a conventional conditional construct

$$S = \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2$$

may have the following performance:

$$\text{Let } T_1 = T(B) + T(S_1) + 2 \times T(\text{then/else})$$
$$= [t_{11}, t_{12}], \text{ and}$$
$$T_2 = T(B) + T(S_2) + T(\text{then/else}) = [t_{21}, t_{22}],$$

where $T(\text{then/else})$ are the control flow times within S, e.g., the time to transfer around S_2 (assuming execution of S_1) or the time to transfer to (or not transfer to) S_2 after testing B. Then $T(S)$ may be simply

$$[\min(t_{11}, t_{21}), \max(t_{12}, t_{22})]$$

(We develop this further in Section III-B.)

The schemas illustrated above for $T(S)$ are independent of the program context of S. Stronger bounds may be derivable when the context is known.

D. Hoare Logic with Time

Standard Hoare logic [12] uses assertions P and Q, respectively before and after a statement S, with the notation

$$\{P\}\ S\ \{Q\}.$$

The interpretation is: if P is true before the execution of S and S is executed, then Q will be true after S (assuming that S terminates). With perfect knowledge of timing, we would augment the above form to:

$$\{P\} < S;\ rt := rt + t(S) > \{Q\}$$

where P and Q could now include relations involving real-time (rt) before and after execution of S, i.e., at the start and end events, respectively, of S, and the brackets ($< >$) indicate that the execution of S and incrementing of rt occur at the same time. (It is assumed that the statement incrementing rt takes zero time.) The axiom of assignment can then be used in P and Q for assertions about rt. For example, if $P = P(rt, \cdots)$, then one can assert either

$$\{P(rt, \cdots)\}\ S\ \{P(rt - t(S), \cdots)\}$$

or

$$\{P(rt + t(S), \cdots)\}\ S\ \{P(rt, \cdots)\}.$$

Our approximation to this unrealizable ideal is the rule:

$$\{P\} < S;\ RT := RT + T(S) > \{Q\}$$

where RT is a pair $[rt_{\min}, rt_{\max}]$ such that at any time, the perfect real-time rt is in the interval defined by $rt_{\min}$ and $rt_{\max}$. P and Q may now include assertions about the elements of RT. Finally, to avoid naming conflicts on RT when statements are composed, we introduce a local real-time variable RT_0 for each statement and use

$$\{P\} < RT_0 := RT;\ S;\ RT := RT_0 + T(S) > \{Q\}. \quad (*)$$

(This trick was employed in [10].) For brevity, we shorten this to the standard Hoare form; henceforth, whenever $\{P\}\ S\ \{Q\}$ appears, it is assumed that this is an abbreviation for (*). Our bounded intervals are exactly the *interval numbers* defined and analyzed in [19], and we use the same kind of arithmetic. For example, if ϕ is one of the operator symbols $+$, $\times$, or $-$, then

$$[a, b]\ \phi\ [c, d] = \{x\ \phi\ y : x \text{ in } [a, b],\ y \text{ in } [c, d]\}.$$

Bounded intervals, such as RT and $T(S)$, will also be treated as sets of numbers. The notation t *in* RT or t *in* $T(S)$ will be used to indicate that t is in the range defined by the bounds. Bounded intervals will be denoted by upper case names.

Examples: The extended logic can express conveniently basic timing properties and constraints. Examples in later sections provide detailed reasonings and proofs for particular program instances.

1) Performance Specifications: If S starts executing in the timing interval RT_{start}, it will finish sometime in the interval $RT_{\text{start}} + T(S)$, i.e.,

$$\{RT = RT_{\text{start}}\}\ S\ \{RT = RT_{\text{start}} + T(S)\}.$$

A simple variation is:

$$\{rt = t_{\text{start}}\}\ S\ \{rt = t_{\text{start}} + t,\ t \text{ in } T(S)\}.$$

This last assertion will be abbreviated sometimes to

$$RT = t_{\text{start}} + T(S)$$

where t_{start} is itself a shortening of $[t_{\text{start}}, t_{\text{start}}]$.

2) Deadlines: Let $RT_{dl} = [t_{dl\min}, t_{dl\max}]$ be deadlines such that a program S must be completed no *earlier* than $t_{dl\min}$ and no *later* than $t_{dl\max}$. This general deadline problem can be expressed (after employing the axiom of assignment):

$$\{RT + T(S) = RT_{dl}\}\ S\ \{RT = RT_{dl}\}$$

i.e., at the start event of S, real-time must be bounded

$$t_{dl\min} - t_{\min}(S) \le rt \le t_{dl\max} - t_{\max}(S).$$

It is impossible to meet this constraint if

$$t_{dl\min} - t_{\min}(S) > t_{dl\max} - t_{\max}(S).$$

From the above (letting $t_{dl\min} = -\infty$) or directly, it is easy to specify a conventional deadline constraint:

$$\{rt \le t_{dl\max} - t_{\max}(S)\}\ S\ \{rt \le t_{dl\max}\}.$$

A similar expression is produced for the earliest finish time constraint given by $t_{dl\min}$.

3) Control of a Periodic Process: Consider a program S which is to be executed periodically, starting at time rt_{start}. An appropriate *invariant* involving the time rt before and after each execution is

$$\text{next} = rt_{\text{start}} + n \times \text{period} = rt + \text{delta}, \ |\text{delta}| \le \text{eps}$$

where next is the computed start time for each cycle (next is usually a program variable), n gives the number of executions of S, period is the time interval allocated to a cycle, and delta is the error, bounded by eps, of next relative to real-time. $T(S)$ is not used directly here, but it is assumed that a delay is introduced in S (using a computer clock) to fill up the entire period. An implementation of the example is developed and analyzed in Section IV.

III. Reasoning and Execution Times for Sequential Programs

A. Expressions and Simple Statements

Consider straight-line programs consisting of assignment statements and procedure calls of the form, respectively

$$v := \text{exp} \quad \text{and} \quad \text{P\_name}(e_1, \cdots, e_n)$$

where v is a variable name, exp is an expression, P_name is a procedure identifier, and $e_1, \cdots, e_n$, $n \ge 0$, are the actual parameters of the procedure call. Assume that exp has the syntax

$$\text{exp} ::= v \mid c \mid \text{F\_name}(e_1, \cdots, e_n) \mid (\text{exp}\ \phi\ \text{exp})$$

c denotes a constant, F_name is a function identifier, and ϕ is a basic binary operator.

A separate analysis of the system, for example, of the compiler and target machine, is done to produce execution times $T(S)$ for primitive entities S as follows:

$T(x)$	Retrieve the value of a variable or constant x. This may just be the time to load a register.
$T(.v)$	Obtain the address of a variable v. In those cases where addresses are known at compile time, $T(.v) = [0, 0]$.
$T(:=)$	Perform an assignment. Typically, the value is the "store" time of the target machine.
$T(\phi)$	Execute the basic operation ϕ.
$T(\text{call/return})$	Call and return from a procedure or function.
$T(\text{par})$	Pass a parameter to a procedure or function. There may be several variations of $T(\text{par})$ according to the type of parameter, for example $T(\text{par\_call\_by\_reference})$ and $T(\text{par\_call\_by\_value})$.

Times for the higher level constructs are computed by adding bounds in the obvious manner. The schema for an assignment statement is

$$T(v := \text{exp}) = T(.v) + T(:=) + T(\text{exp}).$$

A procedure call has the time property

$$T(\text{P\_name}(e_1, \cdots, e_n)) = T(\text{call/return}) + n \times T(\text{par}) + T(\text{P\_body})$$

where P_body is the program text of the procedure. If a parameter e_i is a general expression, then it is necessary to add a corresponding term $T(e_i)$.

These schemas, of course, do not take into account various compiler optimizations that might occur. For example, in the procedure call

$$P1((i + j) \textbf{ mod } k, (i + j) \textbf{ mod } k)$$

a smart compiler might choose to evaluate the common expression only once rather than twice giving execution time bounds

$$T(\text{call/return}) + 2 \times T(\text{par}) + T((i + j) \textbf{ mod } k) + T(\text{P1\_body})$$

assuming both parameters are "value" parameters.

Expression evaluation is handled similarly:

$$T((E_1\ \phi\ E_2)) = T(E_1) + T(\phi) + T(E_2).$$

For example, $T((i + j) \textbf{ mod } k) = T((i + j)) + T(\textbf{mod}) + T(k) = T(i) + T(+) + T(j) + T(\textbf{mod}) + T(k)$.

The sequential composition of statements yields execution times as discussed earlier in Section II-C.

Target machine details, resource allocation strategies for fast registers, and possible compiler optimizations all increase the difficulty of obtaining good bounds manually. However, it is anticipated that machine assistance would be available for analyzing the actual compiler-produced code, especially for the straight-line cases considered in this section.

Example: Let us analyze the simple assignment statement:

$$c := a + b.$$

Using the schema presented here, we obtain the execution time

$$T(.c) + T(:=) + T(a) + T(+) + T(b).$$

This corresponds to a generous machine language program:

load a in $R1$

load b in $R2$

add $R2$ to $R1$

load $.c$ in $R2$

store $R1$ in $R2$ indirect.

But the machine architecture might permit a much tighter program:

load a in $R1$

add b to $R1$

store $R1$ in c

i.e., $T(.c) = T(b) = [0, 0]$ for this instance.

Even better, it may be that the value of a had previously been loaded in a register, reducing the above program to only two instructions. Of course, the other extreme can also happen; registers may contain valuable information which must be saved before executing the assignment statement, thus adding instructions.

For simple cases of straight-line code, it may be that the exact execution time $t(S)$ is predictable. More likely are cases where only bounds will be predictable. The compiled code, for example, may have to compute addresses of variables at run-time. Hardware features, such as memory contention, instruction look-ahead, and caching of results, can also account for a range of possible execution times.

B. Conditional and Looping Constructs

Typical conditional constructs are the if/then/else and case statements:

a) **if** B_1 **then** S_1 **else if** B_2 **then** S_2 **else** $\cdots$ **else if** B_n **then** S_n

b)
case expr **of**
$i_1 : S_1$;
$i_2 : S_2$;
:
$i_n : S_n$
end case $(n \geq 1)$

Conventional realizations for these statements are as follows.

a) if/then/else:
$< B_1 >$
transfer on false to 2
$< S_1 >$
transfer to $n+1$
2: $< B_2 >$
transfer on false to 3
$< S_2 >$
transfer to $n+1$
3: :
n: $< B_n >$
transfer on false to $n+1$
$< S_n >$
$n+1$: {start of next statement}

$< X >$ means the code for construct X.

b) case:
{assumes expr $= i_k$ for some $k \leq n$}
$< x := \text{expr} >$
transfer to x
i_1: $< S_1 >$
transfer to i_{n+1}
i_2: $< S_2 >$
transfer to i_{n+1}
:
i_n: $< S_n >$
i_{n+1}: {start of next statement}

The primitive entities for these statements are the control flow objects as realized by the transfer instructions. Let the time bounds for these objects be given by $T(\text{then/else})$ and $T(\text{case})$. (We assume that all transfers have the same time bounds, so that conditional and unconditional transfers take the same time and $T(\text{then/else}) = T(\text{case})$—but, in general, they may have different values.) Bounds for execution times are then found as follows.

a) if/then/else:

$$\text{Let } T_k = T(S_k) + \min(k+1, n) \times T(\text{then/else}) + \sum_{i=1}^{k} T(B_i) = [t_{k1}, t_{k2}]$$

Then

$$T(\textbf{if } B_1 \textbf{ then } S_1 \textbf{ else } \cdots \textbf{ then } S_n) = \left[\min\left(\min_{k<n} (t_{k1}), n \times T(\text{then/else}) + \sum_{i=1}^{n} T(B_i) \right), \max_{k} (t_{k2}) \right].$$

b) case:

$$\text{Let } T_k = T(\text{expr}) + a_k \times T(\text{case}) + T(S_k) = [t_{k1}, t_{k2}]$$

where $a_k = 2$ for $k < n$ and $a_k = 1$ for $k = n$. Then

$$T(\textbf{case expr of } i_1 : S_1 \cdots S_n \textbf{ end case}) = \left[\min_k (t_{k1}), \max_k (t_{k2}) \right].$$

Because of the detail and tediousness of actually making the calculations, it is expected that machine aids will be available to assist in the analysis of these statements.

To illustrate the schemas and method of analysis for loops, we consider two common instances:

a) an "infinite" loop, usually implementing a cyclic process
loop S **end loop**
b) the classical while statement
while B **do** S **end while**

The primitive loop time, designated $T(\text{loop})$, will just correspond to the time to unconditionally branch back to the beginning of the loop after each execution of S. Similarly, the while statement has a basic object with execution time $T(\text{while})$, corresponding to either the unconditional branch back after execution of S or the conditional test and branch after evaluating B. Like the if/then/else construct, it is assumed that conditional and unconditional branches, regardless of whether or not a branch is taken, all consume the same amount of time—or at least have the same time bounds. (These times could, of course, have different values, but this would not change the methods of analysis, only some of the details.)

For analysis purposes, an auxiliary counting variable is maintained (fictitiously) in every loop as follows:

a) $n := 0$; **loop** S; $n := n+1$ **end loop**
b) $n := 0$; **while** B **do** S; $n := n+1$ **end while**

The counting variable n gives the number of execution of S. The statements involving n take zero time to execute.

It is now straightforward to derive a basic timing *invariant* for both kinds of loops. Let RT_{start} denote the real time bounds at the beginning, i.e., the start event, of either loop. Then, for the infinite loop, we have the following invariant at the start event of S:

a) $RT = RT_{\text{start}} + n \times (T(S) + T(\text{loop}))$
$(n \times [a, b]$ is defined as $[n \times a, n \times b])$.

The while statement is slightly more complex, since B must be tested each time through the loop and a control decision is made whether to continue or transfer out. Taking all of this into account, we obtain the following invariant at S for the while loop:

b) $RT = RT_{\text{start}} + (n + 1) \times (T(B) + T(\text{while})) + n \times (T(S) + T(\text{while}))$

which simplifies to

$$RT = RT_{\text{start}} + (n + 1) \times T(B) + n \times T(S) + (2n + 1) \times T(\text{while}).$$

The same type of analysis leads to execution bounds for the while statement. If, somehow, the number n of executions of the loop is known, then $T(\textbf{while } B \textbf{ do } S \textbf{ end while})$ is just $RT - RT_{\text{start}}$ in the invariant b) above, giving

$$T(\textbf{while } B \textbf{ do } S \textbf{ end while}) = (n + 1) \times T(B) + n \times T(S) + (2n + 1) \times T(\text{while}).$$

We propose to obtain bounds for n using Hoare logic and techniques similar to those developed for proving termination of loops. Let $N = [n_{\min}, n_{\max}]$ and n in N. The timing bounds for the while construct are then

$$T(\textbf{while } B \textbf{ do } S \textbf{ end while}) = (N + 1) \times T(B) + N \times T(S) + (2N + 1) \times T(\text{while})$$

where the notation is defined: $[a, b] \times [c, d] = [ac, bd]$.

At this point, it is worth noting that in order to be able to predict timing behavior, one simply *must* have bounds on such loop executions. There is at least one real-time programming language that explicitly includes limits on either the number of iterations or the execution time of each loop [14].

Examples

1) These ideas are illustrated first with a standard "toy" example, a program to compute an integer approximation to the square root of an integer. Conventional assertions appear as comments in braces.

$\{x \geq 0\}$
$a := 0;$
$\{a^2 \leq x \text{ and } x \geq 0\}$
while $(a + 1)^2 \leq x$ **do** $a := a + 1$ **end while**
$\{a^2 \leq x, x \geq 0, \text{ and } (a + 1)^2 > x\}$ $\{a = \lfloor \sqrt{x} \rfloor\}$

The loop invariant is $\{a^2 \leq x \text{ and } x \geq 0\}$, which upon loop termination, permits the implication $\{a = \lfloor \sqrt{x} \rfloor\}$, i.e., a is the largest integer less than or equal to the square root of x. Here, the auxiliary counting variable n would have the identical values as the program variable a; therefore at termination $n = \lfloor \sqrt{x} \rfloor$. Termination is proven (trivially) by noting that a increases each time through the loop but is also bounded above by $\sqrt{x}$ (i.e., $a^2 \leq x$), which cannot remain true forever. If the input x is restricted to the range $[0, x_{\max}]$, then $N = [0, \lfloor \sqrt{x_{\max}} \rfloor]$ is the tightest possible bounds. A practical one might be $N = [0, \lceil x_{\max}/2 \rceil]$.

Assuming $T(a := 0) = 1$ (i.e., $[1, 1]$), $T(\text{while}) = 1$, $T((a + 1)^2 \leq x) = 4$, and $T(a := a + 1) = 3$, the program has execution time bounds.

$$1 + (N + 1) \times 4 + N \times 3 + (2N + 1) \times 1 = 9N + 6$$

2) A more interesting example is the following program S, presented in [10], which computes $x = \max(1, a, b)$ in a roundabout fashion.

$x := a;$

if $x < 1$ **then** $x := 1;$

while $x < b$ **do** $x := x + 1$ **end while**

Including our auxiliary loop count variable n, the while statement has the invariant

$$\{x \geq 1, n < x, b > 1 \Rightarrow n \leq b - 1,$$
$$b \leq 1 \Rightarrow n = 0\}$$

This invariant provides upper and lower bounds on the number of times the loop statement is executed as a function of b. (Note that this particular invariant does not permit proving that $x = \max(1, a, b)$.)

Let input b be restricted so that $b \leq b_{\max} > 1$; therefore the loop is executed no more than $b_{\max} - 1$ times. Assuming $T(x := a) = 2$, $T(x < 1) = 2$, $T(\text{then/else}) = 1$, $T(x := 1) = 1$, $T(x < b) = 2$, $T(\text{while}) = 1$, and $T(x := x + 1) = 3$, execution times for S can be bounded:

$$\begin{aligned} T(S) &= \big[2 + (2 + 1) + (2 + 1), 2 + (2 + 1 + 1) \\ &\quad + \big(b_{\max} \times 2 + (b_{\max} - 1) \times 3 \\ &\quad + \big(2 \times (b_{\max} - 1) + 1 \times 1\big)\big] \\ &= [8, 7b_{\max} + 2] \end{aligned}$$

IV. Time-Related Statements

A. Higher-Level Constructs for Dealing with Time

Most systems provide a clock abstraction in the form of software functions to read time and possibly also to set the time (e.g., [1], [7], [23]). Define these functions as follows:

get_time	Returns the current value ct of computer time.
set_time(t)	Sets computer time to t (i.e., $ct := t$).

Another basic timing facility, that is useful for programming real-time applications, delays a process for a specific interval of time or until time reaches some absolute value. These are defined:

int_delay(t)	wait until $ct = ct_0 + t$, $t \geq 0$, before proceeding, where ct_0 is the time of the delay statement invocation.
abs_delay(t)	Wait until $ct = t$ before proceeding; proceed immediately if $t \leq ct$.

For "practical" reasons, the delay statements are usually defined more loosely to guarantee continued execution only at some future time $ct \geq ct_0 + t$ and $ct \geq t$, respectively. We give a tighter but still practical specification that permits better timing predictability in the next section.

A third class of timing facility that is commonly available in concurrent systems is the timeout. These are associated with communications and synchronization constructs, and permit the specification of a timing limit when waiting for some event to occur. Timeouts are treated in Section V.

These constructs permit the implementation of more elaborate and higher-level statements and abstractions. One example is the calendar package of Ada, which allows the programmer to perform arithmetic on variables of type "time" in a straightforward manner, and to construct elements of type time from, and to decompose time into, its various components. The Pearl language provides mechanisms for directly scheduling processes based on time [25]. For example, tasks can be activated or resumed at a given time frequency, at a particular time, or after a specified time interval. A third example is the realization of different clocks on the same system, with differing granularities for tick time and even variable interval ticking [21].

B. Timing Analysis of Get_Time and Delay

Given the execution times for standard statements in sequential programs, a straightforward analysis can be made to determine whether a program satisfies some performance constraints or *implicit* timing constraints. However, a principal purpose of the timing-related statements is to program these constraints directly, i.e., explicitly. In order to obtain any sort of sensible predictability, it must be possible to produce tight bounds for these statements. Most languages, even so-called real-time languages such as Ada, do not provide these timing bounds. An important but reasonable question is: does the sequence

int_delay(3); int_delay(2);

produce the same result as the single statement

int_delay(5)

and what *exactly* is the result? (This particular problem is presented in [4], where it is indeed handled precisely in their interesting, but quite different ESTEREL language.)

Consider first some possible realizations of get_time under our initial architectural assumptions (Section II-B). Because computer time is maintained by a separate processor, its value ct will be returned either through a message passing communication or directly if it is accessible through a shared memory. There may be some computation to put ct into a more suitable form. The function could be implemented in several different ways, for example, as a procedure or as an in-line macro. All of these factors must be included to yield the time bounds $T(\text{get\_time})$, which we assume are found by a combination of measurement and analysis.

If $T(\text{get\_time}) = 0$, then the value ct returned would be related to real-time rt as follows:

$$CT = RT + E$$

where ct in CT, rt in RT, and $E = [-\epsilon, \epsilon]$ (ϵ is defined in Section II-A) i.e., ct in $RT + E$. However, in the realistic cases, the ct value used is obtained sometime after the start event of get_time and before its end event. At the end event of get_time, we will therefore be able to bound ct according to the relation.

$$CT + RT + [-t_{\max}(\text{get\_time}), 0] + E$$

where $t_{\max}(\text{get\_time})$ is the upper bound in $T(\text{get\_time})$. Alternatively, ct can be expressed as a function of the real-time at the start event of get_time:

$$CT = RT + [0, t_{\max}(\text{get\_time})] + E.$$

Some definitions and an analysis of the int_delay version of the delay statement are now presented. One reasonable implementation, under our separate processor assumption, might use the get_time function in a busy-wait loop:

a) int_delay(t):

```
Wakeup_Time := get_time + t; {Call this
  statement S1.}
while get_time < Wakeup_Time do {null
  statement} end while
```

Alternatively, a hardware interval timer, *hit*, could be employed directly. It is assumed that the timer decrements its value at each hardware tick and interrupts an applications processor when it reaches zero.

b)
```
hit := t; {Load hit with t.} {S1}
while not hit_interrupt do {null} end while
```

To determine execution time bounds, which also leads to a precise definition of int_delay, we first examine the no overhead case. The ideal, but unrealizable, semantics and time bounds would of course be:

$$T(\text{int\_delay}(t)) = t.$$

However, since delay is implemented with some type of computer clock, we can never do better than

$$T(\text{int\_delay}(t)) = [\max(0, t-\epsilon), t+\epsilon].$$

If $t \geq \epsilon$, a simpler relation is obtained

$$T(\text{int\_delay}(t)) = t + E.$$

For convenience we will assume that $t \geq \epsilon$, but the results for the more general case should also be evident.

Next, consider the effects of various overheads in the implementations given in a) and b) above. We will show that in each case, one can derive useful execution time bounds for the overhead, T_{oh}, that are independent of the parameter t. The final result is

$$T(\text{int\_delay}(t)) = t + E + T_{oh} \text{ (for } t \geq \epsilon).$$

For the code in a), a lower bound of T_{oh} is the sum of the lower bounds of $T(x <$ Wakeup_Time) and $T(\text{while})$, where x is a simple variable or constant. This corresponds to the case where ct has changed to $ct \geq$ Wakeup_Time just prior to the retrieval of ct in get_time.

Analogously, if ct has changed to $ct \geq$ Wakeup_Time just after ct was retrieved in the get_time call in the test, we can trace this worst case scenario to obtain an upper bound of T_{oh} equal to twice the sum of the upper bounds of $T(\text{get\_time} <$ Wakeup_Time) and $T(\text{while})$ plus the upper bound of $T(\text{while})$. A competitor for the upper bound may be the case for small t, when get_time $+ t$ would fail the test even before the execution of S1 has terminated; here the upper bound for T_{oh} is the sum of the upper bounds of $T(\text{S1})$, $T(\text{get\_time} <$ Wakeup_Time), and $T(\text{while})$. The final upper bound for T_{oh} is the larger of these two possibilities.

The code in b) admits to a much simpler analysis. The entire while loop would realistically be coded as a single instruction "branch to self." The while loop of a) would most likely be optimized to eliminate the unconditional transfer at the end of the loop, also. The overhead bounds are

$$T_{oh} = [t_{\min}(\text{inter}), t_{\max}(\text{inter}) + \max(t_{\max}(\text{S1}), t_{\max}(\text{while}))]$$

where $[t_{\min}(\text{inter}), t_{\max}(\text{inter})]$ are the execution bounds for handling the timer interrupt. The handling may be a simple transfer instruction. A "branch to self" is assumed for realizing the while loop. The upper bound considers two cases similar to those treated for a).

The results of this section can now be used to compare the sequence "int_delay(3); int_delay(2)" with int_delay(5). The sequence of two delays has the execution times and meaning:

$$3 + E + T_{oh} + 2 + E + T_{oh} = 5 + 2E + 2T_{oh}$$

while

$$T(\text{int\_delay}(5)) = 5 + E + T_{oh}.$$

Thus, the two are not identical with our timing semantics. (ESTEREL produces identical results for the two in their model because they assume no overhead, instantaneous computations, and no distinction between computer time and real-time).

C. A Program Proof Including Time

A common method for programming a periodic process P is through an (apparently) simple delay loop as illustrated below. P is executed approximately every "period" units of time.

```
next := get_time;
loop
   {The next execution of P starts here.}
   next := next + period;
   x := next - get_time;
   int_delay(x)
end loop
```

A variation of this code, for example, appears in the *Ada Reference Manual* [1].

The code for P could be directly inserted after the comment, or a "start" message might be sent to a separate process implementing P. This schema could also be used to build a higher level software clock that ticks at every "period"; for example, P might then compute the time-of-day.

Informally, we wish to show that the code does indeed compute the correct time for each cycle of P provided that x is always greater than zero; i.e., the variable next always contains the start time of the next or current cycle. As indicated earlier in Example 3 of Section II-D, this requirement can be formalized as a loop invariant I that is always true just after **loop**; I refers to an auxiliary counting variable n (Section III-B):

$$I\text{: next} = rt_{\text{start}} + n \times \text{period and } RT = \text{next} + \text{EPS}.$$

rt_{start} is approximately the program start time, and $EPS = [eps_{\min}, eps_{\max}]$, where $|eps_{\min}|, |eps_{\max}| \leq eps$ for some constant eps.

The problem is first to bound rt_{start} with respect to the real-time at the beginning of the program, say rt_0 (rt_0 is the time at the start event of the first statement). The second, more difficult and more interesting, problem is to find EPS. The program is (supposedly) written so that timing errors do not accumulate each time through the loop; that is the reason for using int_delay (next − get_time) rather than simply int_delay(period). Proving the invariant confirms that this intent has been realized correctly.

Consider the first statement of the program, including the initialization of the fictitious loop count variable. Using our logic, we add provable assertions before and after:

$n := 0$; {Initialize auxiliary counting variable}
{Real-time is assumed equal to rt_0 here}
next := get_time;
{next in $rt_0 + [0, t_{\max}(\text{get\_time})] + E$ (Section IV-B) and $RT = rt_0 + T(\text{next} := \text{get\_time})$}

From the postcondition, next $= rt_0 + e_0'$, where e_0', where e_0' in E' ($E' = [0, t_{\max}(\text{get\_time})] + E$).

Therefore, next $= rt_{\text{start}} + n \times$ period, where $rt_{\text{start}} = rt_0 + e_0'$. Similarly, from $RT = rt_0 + T(\text{next} := \text{get\_time})$, we obtain

$$RT = rt_{\text{start}} - e_0' + T(\text{next} := \text{get\_time})$$
$$= \text{next} - e_0' + T(\text{next} := \text{get\_time}) = \text{next} + \text{EPS}.$$

EPS can be derived from e_0' and $T(\text{next} := \text{get\_time})$; a similar derivation is described later in this section. The postcondition of the first statement thus implies

$$\text{next} = rt_{\text{start}} + n \times \text{period and } RT = \text{next} + \text{EPS}$$

which is our invariant I. We have also bounded rt_{start} with respect to rt_0.

A bound for EPS is produced by proving the invariant in the loop part of the program, again assuming that x is always appropriately greater than zero (see discussion below). We will use the alternate, but equivalent, notation $rt = \text{next} + e$, e in EPS, rather than $RT = \text{next} + \text{EPS}$; and show that e is cancelled each time through the loop, but a one-time residual error is picked up (and cancelled the next time through). The minimum and maximum of this residual error (due to statement times and timing errors) and the initial EPS bounds from above are the desired bounds.

loop
{I}
next := next + period; {Call this statement S1.}
{next $= rt_{\text{start}} + (n+1) \times$ period, $rt =$ next − period $+ s_1 + e, s_1$ in T(S1), e in EPS}
$x :=$ next − get_time; {S2}
{next $= rt_{\text{start}} + (n+1) \times$ period, $rt =$ next − period $+ s_1 + e + s_2$, s_2 in T(S2), $x =$ next − get_time}
{$x = rt_{\text{start}} + (n+1) \times$ period − (next − period $+ s_1 + e + e_0'$), e_0' in E'
$=$ period $- s_1 - e - e_0'$}
$n := n+1$; {Increment auxiliary counting variable.}
{$x =$ period $- s_1 - e - e_0'$, next $= rt_{\text{start}} + n \times$ period, $rt =$ next − period $+ s_1 + e + s_2$}
int_delay(x); {S3}
{next $= rt_{\text{start}} + n \times$ period, $x =$ period $- s_1 - e - e_0'$, $rt =$ next − period $+ s1 + e + s_2 + x + e_0 + h$, e_0 in E, h in T_{oh}} {It is assumed that $x \geq \epsilon$.}
{$rt =$ next − period $+ s_1 + e + s_2 +$ period $- s_1 - e - e_0' + e_0 + h$
$=$ next $+ s_2 - e_0' + e_0 + h$}
end loop {$rt =$ next $+ s_2 - e_0' + e_0 + h + p$, p in T(loop), next $= rt_{\text{start}} + n \times$ period}

After S2, the details of the assertion for x are derived using primarily the results of the get_time analysis in Section IV-B. It is assumed that the start event of S2 corresponds to the real time rt at the start event of get_time, so that the "computer" time returned by get_time is equal to $rt + e_0'$. (An alternative realization would result in the addition of another bounded error term.) The rest of the expression is obtained by substituting for rt and next. After S3, real-time is updated using the results of our previous analysis of int_delay; substituting for x, and assuming $x \geq \epsilon$ every time through the loop, we arrive at the assertion just before **end loop**. The final assertion adds in the effects p of transferring back to the beginning of the loop.

Each time through the loop, we have a new error term $del_1 = s_2 - e_0' + e_0 + h + p$. The first time after initialization gives an error $del_0 = -\delta + \gamma$, where δ in E' and γ in $T(\text{next} := \text{get\_time})$. For each of these, the expression for real-time is of the form:

$$rt = \text{next} + a - b$$

where a in $A = [a_1, a_2]$, b in $B = [b_1, b_2]$. With a little analysis, this yields

$$RT = \text{next} + \text{EPS}$$

with $\text{EPS} = [a_1 - b_2, a_2 - b_1]$. ($\text{EPS} = A - B$ if we define $A - B = [a_1 - b_2, a_2 - b_1]$).

Two pairs for EPS can be found in this manner:

$\text{EPS}_0 = [\text{eps}_{01}, \text{eps}_{02}]$ (from del_0)
and
$\text{EPS}_1 = [\text{eps}_{11}, \text{eps}_{12}]$ (from del_1)

Our final EPS is then

$$\text{EPS} = [\min(\text{eps}_{01}, \text{eps}_{11}), \max(\text{eps}_{02}, \text{eps}_{12})]$$

The analysis does not change substantially if either the execution time of process P or the communication time to P is included. If this time is in $T(S)$, say, then x would receive a value equal to period $- s_1 - e - e_0' - s$, s in $T(S)$. Provided that $x \geq \epsilon$, the invariant would continue to hold. However, if $x < \epsilon$, the system may not work correctly since the timing errors could then accumulate without bound. This, of course, happens when the overhead and/or $T(S)$ is too large for the period; i.e., whenever $\text{period} + \epsilon < s_1 + e + e_0' + s$, where $e = del_0$ or $e = del_1$.

We have shown how the intent of this common code fragment for controlling a periodic process in real-time can be formalized and verified. The example also illustrates the desirability of some machine assistance to keep track of assertions, to verify details (a proof checker), and to compute time bounds.

V. Synchronization and Communications

The objective is to predict execution times for synchronization and communications software in concurrent systems. Statements and mechanisms that need to be analyzed include semaphores, locks, monitors, events, message passing, input–output, remote procedure call, and rendezvous. Process waiting time, i.e., the interval during which a process is logically blocked, is explicitly given for statements such as the int_delay treated in the last section. However, the statements being considered here do not have explicit waiting times, making it more difficult to obtain useful time bounds.

One method is to study the specific context within which one or more of these statements are being used, taking into account the usual overhead times and the details of other processes that interact with the one in question. As an example, consider the following standard producer-consumer system, consisting of two processes that communicate through a bounded buffer and use semaphores for synchronization (e.g., [5]):

```
producer process:
  loop
    Produce_Next_Record;
    P(empty); Fill_Buffer; V(full)
  end loop

consumer process:
  loop
    P(full); Empty_Buffer; V(empty);
    Consume_Record
  end loop
```

Initially, the semaphore empty $= n$, the number of buffers, and full $= 0$.

Let us find $T(P(\text{empty})) = [t_{e\min}, t_{e\max}]$. Assuming that a successful P operation has the predetermined bounds $T(P_{\text{success}}) = [tp_{\min}, tp_{\max}]$, then $t_{e\min} = tp_{\min}$. $t_{e\max}$ is derived by noting that in the worst case, the producer must wait at $P(\text{empty})$ for an entire cycle of the consumer, starting at Consume_Record, before the $V(\text{empty})$ occurs to wake it up. This yields an upper bound $t_{e\max}$ equal to the upper bound of the expression:

$$T(P_{\text{fail}}) + T(\text{Consume\_Record}) + tp_{\max} + T(\text{Empty\_Buffer}) + T(V_{\text{wakeup}})$$

where $T(P_{\text{fail}})$ is the predetermined bounds for a P operation that fails (i.e., the time up to the blocking of the process), $tp_{\max}$ corresponds to the successful $P(\text{full})$, and $T(V_{\text{wakeup}})$ is the time for a V operation that also wakes up a blocked process. A similar analysis can be made for $T(P(\text{full}))$. Of course, there are many cases of interest where this approach is not practical.

A. Timeouts

A second method that is guaranteed, in principle, to yield bounds employs timeouts. Associated with each invocation of a synchronization or communication operation is a time, say t. If the process is not unblocked by t, then a "timeout" occurs, the operation is completed with an appropriate indication, and the process continues. Alternatively, the timeout parameter could be an interval rather than an absolute time. Timeouts are often used as indicators of various kinds of system failures, such as a hardware error, software mistake, or a deadlock. They are also convenient for programming timing constraints, as illustrated below. One commercial example is the VAX-Elan Toolkit [23] which provides timeouts for waiting on any of input–output devices, semaphores, events, or message ports.

Timeouts in operations may be implemented with the same basic clock mechanisms as used for int_delay. That is, a hardware interval timer or absolute timer could be directly employed, or the get_time function could be invoked in a busy wait loop. In either case, the implementation details are more complex since the timeout must be

coordinated with the primary object or resource that is being requested (message, semaphore, event, · · ·).

Example: Define a semaphore P operation with timeout as follows:

$P(S, t)$ Wait until either $S > 0$ or time $= t$, whichever happens first.
If $S > 0$ then $S := S - 1$ and return true.
Otherwise, return false.
{The operation is assumed to be atomic}.

$T(P(S, t))$ must be some function of both the parameter t and the time of the start event for $P(S, t)$ for a particular call of the statement. The lower bound component is almost certainly always $tp_{\min}$. (A possible exception is mentioned below.) The upper bound represents the worst case time when a timeout occurs. Two cases are possible: an "immediate" timeout because t is too small or a later timeout.

In the first case, $t < t_0$ for some computer time t_0 that is the *first* one compared against t. Execution bounds here are the times for entering P, making the tests against S and t, and returning. (If the lower bound is smaller than $tp_{\min}$, then this also becomes the lower bound for $T(P(S, t))$.)

The failure for the second case occurs at some computer time $ct = rt + \delta$, δ in E, such that $ct = t + e_1$, where e_1 accounts for any missed ticks of ct due to overhead. Therefore, the timeout occurs at real-time $ct - \delta = t + e_1 - \delta$. Adding in a term e_2 for the time to clean up and return after the timeout, we obtain a final event time $rt_f = t + e_1 - \delta + e_2$. The upper bound is the maximum of rt_f less the time of the start event of $P(S, t)$. The maximum of rt_f is $t + \epsilon + e_{\max}$, where $e_{\max}$ is the sum of the maximums of e_1 and e_2.

One purpose for working out the details in this example is to note that bounds should be derivable for each element in these timing expressions, e.g., e_1 or e_2.

B. Communications with Timeouts and Time Stamps

Consider a *synchronous* communications mechanism for handling a simple form of events. An event is posted or sent with a send primitive and is obtained or received with a receive operation. Both the sender and receiver are blocked on their respective primitives until either a rendezvous or a timeout occurs. Events are time stamped in the loose sense that the rendezvous time is also returned at the completion of the operations.

The receive function will be used and defined in the following manner:

ev := receive(event_name, t)

where event_name is an input identifying the type of event, t is an absolute time used as the timeout, and the returned value ev is a pair ev.name and ev.time. If a timeout occurs, ev.name = timeout and ev.time is nominally equal to t. Otherwise ev.name = event_name and ev.time gives the time stamp value, ts say, $ts < t$. ts is some computer time computed sometime between the start and end events of the receive. One could also specify a sender process, a port, or a channel as part of the input of receive, but this is not necessary for our timing analysis purposes. (*Aside:* In the last section, our definition of $P(S, t)$ permitted a true return, i.e., no timeout, if time $= t$; here, for variety, we assume that $ts < t$ on an event.)

In this section, we will analyze receive and a nontrivial example that uses timeouts and time stamps. The send operation will not be examined or defined in any detail, since we will only need it in a peripheral way.

First consider the no-timeout case for receive, when ev.time $= ts < t$; ts is the rendezvous time obtained by invoking, say, get_time, in the receive. The real time corresponding to ts is $ts + \delta$ for some δ in E. Let T_{roh} be the bounds for the software overheads in receive. Then at the end event of ev := receive(event_name, t), the real time is $rt = ts + \delta + e$, where e in T_{roh}.

When the timeout event occurs, computer time $ct \geq t$. Two possibilites arise, similar to those treated in the $P(S, t)$ analysis (Section V-A). If the timeout value t was too small initially, for example, if it was less than computer time at the start event of receive, then the real time at the end event of receive is $rt_{\text{start}} + e$, e in T_{roh}, where rt_{start} is the real-time at the start event of the receive. The second case timeout occurs at computer time $ct = t + e_1$, where e_1 accounts for any missed ticks of ct due to overhead. Real time at the timeout is therefore $t + e_1 + \delta$, δ in E. Finally, real time at the end event of the receive can be expressed as $t + \delta + e$, e in T_{roh}, where we have included e_1 in e as well as the other overheads.

The examples to be analyzed are programs to recognize single and double clicks from a button on a mouse input device (e.g., [6]). A user clicks the button by depressing it (D for button down) and then letting it go (U for button up). If the time between the D and U events is less than some given interval t_{sc}, a single click event is defined. A double click event is two single clicks separated by an interval less than a given t_{dc}; more specifically, $t_{D2} - t_{U1} < t_{\text{dc}}$, where t_{D2} is the time of the second click's down event and t_{U1} is the time of the first click's up event.

A recognizer for single clicks might be the following program:

```
{Recognize a single click.}
loop
   down := receive(D, inf);
                {inf is a very large number.}
   t1 := down.time + tsc;
   up := receive(U, t1);
   if up.name = U then {single click} send (SC)
   else {timeout} receive(U, inf)
end loop
```

Our aim is to show that at the "send(SC)," we have down.name = D, up.name = U, and $rt_{\text{up}} - rt_{\text{down}} < \text{tsc} + \text{err}_1$, where rt_{up} is the real time at the up event, rt_{down} is real-time at the down event, and err_1 is some small error term that can be bounded. Similarly, we wish the follow-

ing assertions to hold at the timeout: down.name = D, up.name = timeout, and $rt_{up} - rt_{down}$ = tsc + err_2, where rt_{down} is as before, rt_{up} is the real-time at the timeout event of receive(U, t1), and err_2 is an error term. Note that this formulation also shows the existence of an ambiguous window around tsc, where single clicks may be detected or missed depending on the relative values of particular instances of err_1 and err_2.

Using our timing logic and the results of the analysis of receive, we can annotate the program with provable assertions:

```
loop
  down := receive(D, inf); {S1}
  {down.name = D, rt_down in down.time + E, rt in down.time + E + T_roh}
  t1 := down.time + tsc; {S2}
  {t1 = down.time + tsc, rt_down in down.time + E,
    rt in down.time + E + T_roh + T(S2)}
  up := receive(U, t1); {S3}
  {up.name = U ⇒ (rt_up in up.time + E, rt in up.time + E + T_roh, up.time < t1),
    up.name = timeout ⇒ (rt in t1 + E + T_roh or
      (rt in down.time + 2 × E + 2 × T_roh + T(S2) and
      rt in t2 + E + T_roh, t2 > t1))}
  if up.name = U then send(SC)
  else received(U, inf)
end loop
```

Since rt_{down} in down.time + E after S1, we have rt_{down} = down.time + δ_1, δ_1 in E. Similarly, for the case that up.name = U after S3, we have rt_{up} = up.time + δ_2, δ_2 in E. The assertions after S2 and S3 give

down.time = t1 − tsc and up.time = t1 − deltat1 for some deltat1 > 0.

Combining these, we get

$$rt_{up} - rt_{down} = \text{up.time} + \delta_2 - \text{down.time} - \delta_1$$
$$= \text{tsc} - \text{deltat1} + \delta_2 - \delta_1$$

Since δ_1, δ_2 in E, the difference can be bounded:

$$rt_{up} - rt_{down} < \text{tsc} + \text{err}_1, \text{ for } \text{err}_1 = 2\epsilon$$

When up.name = timeout, the situation is slightly more complex because there is the possibility that the interval tsc is smaller than the execution time overhead between the down and up clicks. Provided tsc is large enough, the first implication on the timeout assertion holds and we also have

rt_{up} = t1 + δ_1 + e, δ_1 in E, e in T_{roh}.

Using the same expressions for rt_{down} and t1 as before, we get

$$rt_{up} - rt_{down} = \text{t1} + \delta_1 + \text{e} - (\text{t1} - \text{tsc} + \delta_2)$$
$$= \text{tsc} + \delta_1 - \delta_2 + \text{e}, \delta_1, \delta_2 \text{ in } E, \text{ e in } T_{roh}$$
$$= \text{tsc} + \text{err}_2$$

where err_2 in $2E + T_{roh}$.

If tsc is too small, the program will always timeout, indicating that the system is not fast enough to recognize a single click. To be safe, tsc should certainly be larger than the maximum of $T(S2) + 2T_{roh} + 2E$.

A recognizer for a double click employs similar ideas:

```
{Recognize a double click.}
down1 := receive(D, inf);
t1 := down1.time + tsc;
loop
  up1 := receive(U, t1);
  if up1.name = U then {first single click}
  begin
    t2 := up1.time + tdc;
    down2 := receive(D, t2);
    if down2.name = D then {Start 2nd single click.}
    begin
      t1 := down2.time + tsc;
      up2 := receive(U, t1);
      if up2.name = U then {double} send(DC)
      else up2 := receive(U, inf)
    end
  end
  else up1 := receive(U, inf);
  down1 := receive(D, inf);
  t1 := down1.time + tsc
end loop
```

The program logic is sufficiently complex to warrant some arguments in favor of correctness. Assuming that computer time and real time are identical and that overhead time is zero, the following assertions hold at the double click send(DC):

```
First_Single_Click: down1.name = D, up1.name = U, up1.time - down1.time < tsc
Second_Single_Click: down2.name = D, up2.name = U, up2.time - down2.time < tsc
Double_Click: {First_Single_Click, Second_Single_Click,
  down2.time - up1.time < tdc}
```

Similarly, assertions of correctness can be made and proven at the timeout points and at the single click event. We can then add in the overhead and timing error terms to obtain the real-time properties, using the same approach as for the single clicker.

VI. Interrupt Handling and Processor Sharing

Our assumption of dedicated processors (Section II-B) applies to some contemporary systems; it can be expected to hold more often for future applications where timing predictability and simplicity are more important than potential cost savings through resource sharing. Still, there are now, and will continue to be, many systems that must deal with processor sharing problems. Typically, a processor may be shared between a user process and a clock process, between a user process and one or more input–output processes, among several user processes, and combinations of these. An operating system is also required in order to regulate the multiplexing among processes. Timer and input–output driver processes are usually driven by or closely connected to hardware interrupts; the code for handling an interrupt often implements the software part of one of these processes. In this section, we show how the timing behavior for *some* of the above sharing can be predicted.

It is worth noting that the techniques for computing $T(S)$ and the timing logic presented in the previous sections can be used without change for those parts of a system that are *not* interruptible, i.e., where interrupts are masked off. Thus, for example, our methods can be employed to analyze the kernel and many higher-level components of an operating system, that are global critical sections and uninterruptable. Similarly, our techniques apply to segments of applications processes that are run in a no-interrupt mode because of shared data possibilities or the need for predictable performance.

When interrupts are permitted and both interrupt handling times and frequencies can be bounded, the effects of processor sharing between a user process and one or more interrupt-handlers can be included in a timing analysis. The interrupt-handler for a clock interrupt, a "tick," might just update a variable representing time; the driver handler for an input-output device might update some state variables, transfer data to or from a buffer, and initiate another operation.

Suppose that an interrupt handler IH can be characterized by the following parameters:

$F = [f_{\min}, f_{\max}]$	Bounds on the frequency of interrupts (number of times per second).
$T(\mathrm{IH})$	Execution time for the code.

The interpretation is that the interval between interrupts is no smaller than $1/f_{\max}$ seconds and no longer than $1/f_{\min}$ seconds. Let $T(S)$ be the execution time bounds of a statement S running on a dedicated processor and $T'(S)$ be the time bounds for S including processor sharing with the interrupt handler. Then, the upper bound in $T'(S)$ is

$$t'_{\max}(S) = t_{\max}(S) + t'_{\max}(S) \times f_{\max} \times t_{\max}(\mathrm{IH})$$

giving

$$t'_{\max}(S) = t_{\max}(S)/(1 - f_{\max} \times t_{\max}(\mathrm{IH}))$$

A similar result is obtained for $t'_{\min}(S)$. The proof rule (*) of Section II-D is changed in a straightforward way to reflect the new execution times:

$$\{P\} < \mathrm{RT}_0 := \mathrm{RT};\ S;\ \mathrm{RT} := \mathrm{RT}_0 + T'(S) > \{Q\}$$

Note that our results for $T'(S)$ should be adjusted upward so that $t'_{\max}(S) = t_{\max}(S) + k \times t_{\max}(\mathrm{IH})$, for some integer k, to eliminate "fractional" handling of the last interrupt; a similar downward adjustment should be made for $t'_{\min}(S)$. The level of granularity at which these results should be applied depends on F; clearly, one wishes to avoid a build-up of rounding-up and rounding-down errors, that would make the final bounds meaningless. Assuming that interrupt handlers are noninterruptable, the effects of more than one handler can also be incorporated using the same ideas.

The interrupt-handling results can also be employed for the analysis of more elaborate instances of processor sharing. One example is the class of slice-based software architectures in common use in hard real-time systems [3]. Processes are broken into nonpreemptible units called slices; these are scheduled by a cycle executive, driven by a timer, that consults a scheduling table at each clock tick [21]. Each slice can be analyzed for execution times *a priori*, given also some frequency and bounds information on the clock and the input–output interrupt handlers. Slices can be interrupted but not preempted—the slice continues after a handler completes. The cyclic executive itself is a nonpreemptible process whose timing behavior can also be obtained. An extreme but practical example of a slice-based system is one where each cycle of a periodic process is nonpreemptible and thus constitutes a slice (e.g., [2]).

Timing predictability seems impractical when a process can be preempted at arbitrary points in its code. If processes that are sharing a processor use time-related functions such as int_delay, their execution times also appear to be unpredictable in general; for example, if two processes are waiting on an int_delay and are "scheduled" to wakeup at the same time, then it is not evident how to associated practical time bounds with each process's delay. More study is needed to define particular useful cases that can (or cannot) be analyzed.

VII. Discussion

The methodology for specifying, predicting, and proving assertions about time is new and promising, but still incomplete and untested in practice. The major outstanding issue is whether or not *useful* best and worst case execution time bounds can be found for statements in con-

temporary higher-level languages and their underlying systems.

The obvious next step is to perform some experiments with programs in a suitable implemented language/system. The aim would be to determine whether some combination of measurements and analysis of the language, compiler, run-time system, and target architecture will yield good deterministic bounds and also confirm our definitions of the basic timing elements. In order to compute bounds on structured statements in general and to compute times for statements in particular programs, it would be desirable to build some automatic tools to help in the analysis, for example, tools that are realizations of our algorithms for various statement schemas (e.g., Section III-B). Software for doing interval arithmetic is available (e.g., [16]) and should be a part of these tools.

Structured timing schemas for the components of higher-level languages, such as the schemas proposed in the paper, correspond well with the philosophy and straightforward code generation of some modern compilers, e.g., [26]. These seek to improve correctness, understanding, and maintainability by using simple parsing and predictable code generation.

A difficult problem is the determination of tight timing information on instructions and code sequences for contemporary computers; pipelining, caching, and a host of other performance-enhancing features seem to hinder timing predictability. By selecting appropriate granularities for the higher-level language elements, we hope that these and other hardware issues, such as exactly how and where to incorporate worst case effects of memory and bus contention, can be handled practically. This work may also result in a clearer understanding and definition of those machine architectures and hardware features that permit timing predictability (perhaps in a manner similar to that reported in [27] where some microprocessor architectures are evaluated based on their suitability for compiler code generation).

There is also a need to analyze other types of statements, including standard input–output instructions, other forms of synchronization and communications operations, and guarded commands in conditional and iterative statements. Finding and proving the "right" kinds of assertions has been somewhat difficult; further experiences in reasoning with and about time should simplify this problem. Timing assertions and the associated proofs seem to be relatively simple compared with those required for logical correctness, leading to optimism about the viability of our approach.

The main results and contributions of this work are the techniques that, in principle, permit the prediction of the timing as well as the logical behavior of programs. We believe that all of the following are novel: the idea and methods for computing time bounds, the ability to deal directly with both real-time and computer times, the extension and application of Hoare logic for reasoning about time, and the illustration of specific assertions involving time for common problems.

Acknowledgment

I am grateful to P. Chretienne, S. Fdida, C. Girault, and J. Zahorjan for some helpful discussions. Special thanks go to K. Jeffay for a critical and detailed reading of [22]; several of his suggestions have been followed in the revision.

References

[1] *Military Standard Ada Programming Language*, U.S. Dep. Defense, Washington, DC, ANSI/MIL-STD-1815A, Jan. 1983.

[2] L. L. Alger and J. H. Lala, "A real time operating system for a nuclear power plant computer," in *Proc. Real-Time Systems Symp.*, IEEE Comput. Soc., Dec. 1986, pp. 244–248.

[3] T. P. Baker and G. M. Scallon, "An architecture for real-time software systems," *IEEE Software*, pp. 50–58, May 1986.

[4] G. Berry and L. Cosserat, "The ESTEREL synchronous programming language and its mathematical semantics," in *Lecture Notes in Computer Science 197, Seminar in Concurrency*, S. Brooks, A. Roscoe, and G. Winskel, Eds. New York: Springer-Verlag, 1985, pp. 389–448.

[5] L. Bic and A. Shaw, *The Logical Design of Operating Systems*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[6] L. Cardelli and R. Pike, "Squeak: A language for communicating with mice," in *Proc. SIGGRAPH '85, ACM SIGGRAPH*, vol. 19, no. 3, pp. 199–204, July 1985.

[7] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager, "Thoth, a portable real-time operating system," *Commun. ACM*, vol. 22, pp. 105–114, Feb. 1979.

[8] R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze, "Toward real-time performance benchmarks for Ada," *Commun. ACM*, vol. 29, pp. 760–778, Aug. 1986.

[9] B. Dasarathy, "Timing constraints of real-time systems: Constructs for expressing them, methods for validating them," *Commun. ACM*, vol. 29, pp. 80–86, Jan. 1985.

[10] V. H. Haase, "Real-time-behavior of programs," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 454–501, Sept. 1981.

[11] ——, "Modular design of real-time systems," in *System Description Methodologies*, D. Teichroew and G. David, Eds. Amsterdam, The Netherlands: Elsevier North-Holland, 1985, pp. 91–100.

[12] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[13] F. Jahanian and A. K.-L. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 890–894, Sept. 1986.

[14] E. Kligerman and A. D. Stoyenko, "Real-time Euclid: A language for reliable real-time systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 941–949, Sept. 1986.

[15] D. E. Knuth, *The Art of Computer Programming, vol 1/Fundamental Algorithms*, 2nd ed. Reading, MA: Addison-Wesley, 1973.

[16] U. W. Kulish and W. L. Miranker, Eds., *A New Approach to Scientific Computation*. New York: Academic, 1983.

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[18] A. K.-L. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, May 1983.

[19] R. E. Moore, *Interval Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1966.

[20] A. Pnueli, presentation given at the Real-Time Systems Issues Workshop, Univ. Texas, Austin, Dec. 1986.

[21] A. Shaw, "Software clocks, concurrent programming, and slice-based scheduling," in *Proc. Real-Time Systems Symp.*, IEEE Comput. Soc., Dec. 1986, pp. 14–18.

[22] ——, "Reasoning about time in higher-level language software," Lab. MASI, Univ. Paris 6, Res. Rep., Apr. 1984.

[23] *VAXElan Technical Summary*, Digital Equipment Corp., 1983.

[24] R. A. Volz and T. N. Mudge, "Instruction level mechanisms for accurate real-time task scheduling," in *Proc. Real-Time Systems Symp.*, IEEE Comput. Soc., Dec. 1986, pp. 209–215.

[25] N. Werum and H. Windauer, *Introduction to PEARL*, 2nd ed., Berlin, Germany: Vieweg, 1983.

[26] N. Wirth, "A fast and compact compiler for Modula-2," Inst. Informatik, ETH, Zurich, Tech. Rep. 64, July 1986.

[27] ——, "Microprocessor architectures: A comparison based on code generation by compiler," *Commun. ACM*, vol. 29, pp. 978–990, Oct. 1986.

Alan C. Shaw received the B.A.Sc. degree in engineering physics from the University of Toronto, Toronto, Ont., Canada, the M.S. degree in mathematics, and the Ph.D. degree in computer science, both from Stanford University, Stanford, CA.

He has worked as a Systems Engineer with the IBM Corporation, Research Associate at the Stanford Linear Accelerator Center, Visiting Professor at ETH, Zurich, and at the University of Paris, and as a member of the faculty of Computer Science at Cornell University. He first joined the faculty of the Department of Computer Science at the University of Washington in 1971. His current research interests are in operating systems, real-time systems, software engineering, and software specification methods. In addition to other research and scholarly publications, he is coauthor of the textbook *The Logical Design of Operating Systems*, 2nd edition.