Language Design for Reliable Software
S.L. Graham Editor

Toward a Discipline of Real-Time Programming

Niklaus Wirth
Federal Institute of Technology (ETH), Zürich, Switzerland and
Xerox Palo Alto Research Center

Programming is divided into three major categories with increasing complexity of reasoning in program validation: sequential programming, multiprogramming, and real-time programming. By adhering to a strict programming discipline and by using a suitable high-level language molded after this discipline, the complexity of reasoning about concurrency and execution time constraints may be drastically reduced. This may be the only practical way to make real-time systems analytically verifiable and ultimately reliable. A possible discipline is outlined and expressed in terms of the language Modula.

Key Words and Phrases: multiprogramming, real-time programming, process synchronization, processor sharing, program validation, Modula
CR Categories: 3.80, 4.22

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

A version of this paper was presented at the SIGPLAN/SIGOPS/SICSOFT Conference on Language Design for Reliable Software, Raleigh, N.C., March 28-30, 1977.

Author's present address: Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, CA 94304.

Concepts and Terminology

Conventional programs describe actions that change the values of variables in discrete steps. Execution of these actions by any processor takes a finite amount of time. This time is not a characteristic property of the program, but rather of the employed processor. In the interest of generality, programs are usually designed such that the computed results are independent of the execution speed of their processor(s). The obvious way to obtain time independence is to restrict programs to describing purely sequential chains of actions. Each action is initiated when its unique predecessor has been completed. Such a program is called a *sequential program.*

Apart from the commercial necessity of cost accounting, the reasons for introducing the notion of time into the programmers' considerations are purely technical. The principal reason is that computation can be speeded up by the use of several concurrently operating processors. Often processors are needed with special capabilities not shared by others (such as input/output devices). The viable aim is to utilize them as effectively as possible. Programs using several processors consist of several routines, called *processes,* which are themselves purely sequential, are executed concurrently, and communicate via shared variables and synchronization signals. A program that specifies (possible) concurrency is called a *multiprogram.*

It is prudent to extend the conceptual framework of sequential programming as little as possible and, in particular, to avoid the notion of execution time, in spite of the fact that time was the ultimate reason for multiprogramming. Multiprograms should, whenever possible, be designed with sufficient generality that they specify the computed results independently of the absolute and relative speeds of employed processors. The only indispensible assumption is that these speeds are greater than zero. Adherence to execution time independence affords the tremendous advantage that a program's validity can be deduced solely from the static program text containing logical assertions on the state of the computation after each statement and signal exchange. If we depart from this rule and let our programs' validity depend on the execution speed of the utilized processors, we enter the field commonly called *"real-time"* programming. (From the foregoing it appears that "processing-time dependent" programming would be a more descriptive term.)

Which are the reasons for designing execution-time-dependent programs? A serious reason is that certain processes – which are not programmable at discretion, as they may be part of the environment – may fail to wait for synchronization signals indicating completion of the cooperating partner's task. As a result, cooperation with such processes will necessarily have to depend on processor speed. For example, a sensor in a traffic control system emits a signal each time a car passes, but

will not await acknowledgment of signal registration before emitting another signal. Hence validation of the computerized part of the system will depend on, for instance, a lower bound on the interval between two consecutive sensor signals. In other examples, the nonprogrammable partner may well be a human unwilling to wait for the results of his request longer than a certain duration.

The essential point here is that the sets of concepts (for reasoning) and facilities (for description) of real-time programming should be small extensions of those governing multiprogramming, which in turn should be small extensions of those used in sequential programming. Only very few fundamental concepts are added in each of the two steps, namely synchronization signals and mutual exclusion for multiprogramming and execution speed for real-time programming. Yet in each step, the complexity of reasoning about programs has been increased by a new dimension. Only by adhering to a restrictive discipline can validation problems be kept intellectually manageable. Such a discipline is decisively shaped by the programming language used. We have developed the high-level language *Modula* with the aim of making the advantages of reasoning in terms of high-level languages amenable to the realms of multiprogramming and real-time programming [6]. Here we shall illustrate the relevant parts of that experimental language by means of a few examples and attempt to outline a set of rules to cope with real-time phenomena. These rules are still tentative and incomplete. Further research is needed to achieve a full understanding of the subject. But perhaps such rules will ultimately lead to a discipline.

An Example

Consider first the purely sequential process in which repeatedly a data portion is produced and deposited in a buffer, and thereupon fetched and consumed.

```
process ProducerConsumer;
  var buffer: BufferType;
begin
  loop produce; deposit;
    fetch; consume
  end
end ProducerConsumer
```

Now assume that the operations *produce* and *consume* can be executed by two—perhaps different—processors and that we wish to express this possibility explicitly by a multiprogram, taking advantage of concurrency by multiple buffering. Hence we need to express coordination: the producer must wait when the buffer is full, and the consumer when it is empty. Using the synchronization primitives offered by the language Modula, namely shared variables and so-called signals, we can express the program as follows:

```
module ProducerConsumer;
  var n: integer; (*no. of occupied buffer slots*)
    nonfull, nonempty: signal;
    buffer: BufferType; (*N slots*)
  process Producer;
  begin
    loop produce;
      if n = N then wait(nonfull) end;
      (*buffer not full*) deposit;
      n := n + 1; send(nonempty)
    end
  end Producer;
  process Consumer;
  begin
    loop if n = 0 then wait(nonempty) end;
      (*buffer not empty*) fetch;
      n := n - 1; send(nonfull);
      consume
    end
  end Consumer;
begin (*initialize n and start processes*)
  n := 0; Producer; Consumer
end ProducerConsumer
```

This scheme guarantees that the processes synchronize at the appropriate moments and hence execute exactly the same number of *produce* and *consume* operations. But in general the scheme depends on the two statement sequences starting with "if" and ending with "send( )" being *mutually exclusive*.

To ensure this mutual exclusion in a Modula program, the critical statement sequences must be formulated as procedures which are declared in a so-called *interface module*. This module also contains declarations of all shared variables and shields them from external access. Brinch Hansen [1] and Hoare [3], who call the interface module a *monitor*, proposed this scheme. As it ensures that shared variables are accessed under mutual exclusion only, it makes sequential programming verification rules applicable to multiprograms adhering to the scheme's discipline [3, 4].

The define- and use-lists in the heading of a module explicitly define its precise transparency. The define-list indicates the local objects to be visible outside, and the use-list enumerates the nonlocal objects to be visible inside the module. Hence there is no dogmatic rule ensuring safety against every possible blunder (for example, the buffer could also be declared visible to the outside), but instead there are facilities that cater to the recommended discipline and encourage the programmer to adhere to it wherever applicable.

```
module ProducerConsumer;
  type ElementType = . . . ;
  interface module Buffering;
    define Fetch, Deposit;
    use N, ElementType;
    var n: integer; (*no. of filled buffer elements*)
      nonfull, nonempty: signal;
      buffer: BufferType; (*N slots*)
    procedure Fetch(var x: ElementType);
    begin if n = 0 then wait(nonempty) end;
      (*n > 0*) fetch(x); n := n - 1; send(nonfull)
    end Fetch;
```

```
procedure Deposit(x: ElementType);
begin if n = N then wait(nonfull) end;
  (*n < N*) deposit(x); n := n + 1; send(nonempty)
end Deposit;
begin (*initialize buffer management*) n := 0;
end Buffering;
process Producer;
    var y: ElementType;
  begin
    loop produce(y); Deposit(y)
    end
  end Producer;
  process Consumer;
    var z: ElementType;
  begin
    loop Fetch(z); consume(z)
    end
  end Consumer;
begin Producer; Consumer
end ProducerConsumer.
```

Now assume that the producer is designed such that it fails to await the *nonfull* signal. It may, for example, be a card reader initiated to read a card, or a tape unit reading a block of characters. The card (tape), once in motion, cannot be stopped: the 80 characters are "produced" whether or not the consumer is ready to accept them. Under these unsafe circumstances, the scheme works properly if and only if the consumer is always faster than the producer. In the case, the consumer would never have to send its *nonfull* signal, as the producer would be bypassing its *wait* statement if there were one. This, however, implies that we rely on some knowledge of execution speed; we have entered the domain of real-time programming. The assertion *buffer not full* in the producer must now be derived from the premise that it is a consequence of the statement *fetch*, and that

T("fetch; full := false; send( ); consume") < T("produce")

where T(S) denotes the (maximum) time to execute statement S.

A First Conclusion

Program verification is best manageable in the case of purely sequential programs, where assertions on the state of the computation provide an effective handle for keeping track of the effects of each statement. The problem is considerably more complicated in multiprogramming. However, the disciplined use of common variables and synchronizing signals in explicitly designated interface sections with mutual exclusion reduces the added complexity of validation drastically. Such a disciplined use is aided (and can even be enforced) by high-level language features such as monitors and conditions [3] or modules and signals [6]. In effect they make multiprograms which adhere to this discipline amenable to the verification techniques of sequential programs.

Similarly, dependence on execution-time constraints makes the verification task almost intractable unless a discipline is observed that confines time dependence to certain isolated parts in some standard pattern. From the foregoing, my conclusion for making real-time programs manageable can be condensed into the recipe:

1. First formulate the entire program *without* any reliance on execution times. Explicitly provide all synchronization signals needed for this generality.
2. For each signal that is not made available by the machinery to be used, derive analytically the time constraints that allow the absence of the signal.
3. Check whether these constraints are met by your computer system.

The first step constitutes a task in "ordinary" multiprogramming with —more or less—established validation techniques. The second step is relatively straightforward, provided the time-dependent parts of the program are few, simply structured, and without loops with an unknown number of repetitions.

An Interlude: Signals and Semaphores

The above rule may even be useful in situations where a priori the notion of time does not seem to play a role at all. In certain situations programmers make—perhaps unconsciously—assumptions about relative execution speeds of processors. If they consciously refrained from doing so, the programs might become slightly more complicated. Specifically they use *signals* where they should use *semaphores*. Because this mistake does not seem to be uncommon in spite of its disastrous effects, we shall briefly characterize it.

Semaphores have been postulated as synchronization primitives by Dijkstra [2]. Hoare and Brinch Hansen have instead proposed *conditions* [3] and *queues* [1]. We have renamed them *signals* in Modula [6], in analogy to pulse or trigger signals in electronic circuits. Semaphores can be expressed in terms of signals and ordinary variables. Signals must therefore be considered as more primitive entities. We subsequently present a formulation of binary semaphores (bs) and general semaphores (gs) and their P and V operations in terms of Modula (declared as an interface module).

```
type bs =
  record s: signal;
    b: Boolean
  end
procedure P(x: bs);
begin
  if x.b then wait(x.s) end;
  x.b := false
end
procedure V(x:bs);
begin x.b := true; send(x.s)
end
```

```
type gs =
  record s: signal;
    n: integer
  end
procedure P(x: gs);
begin
  if x.n = 0 then wait(x.s) end;
  x.n := x.n - 1
end
procedure V(x: gs);
begin x.n := x.n + 1; send(x.s)
end
```

Evidently the semaphore is equivalent to a signal with associated memory. It remembers whether or not the corresponding signal had been sent (bs), or even counts the number of past unresponded signal emissions (gs).

Now assume that two processes P1 and P2 start executing respective statements S1 and S2 simultaneously and that it is requested that before continuing after S2, P2 wait for the completion of S1 by P1. If we happen to know that S2 is executed (considerably) faster than S1, we are (much) tempted to use the following schema involving a completion signal s:

P1: . . . S1; send(s); . . .
P2: . . . S2; wait(s); . . .

However, if for some—of course not unforeseeable but nevertheless unforeseen—reasons sometimes S2 is not completed before S1, then the signal may be emitted without P2 noticing it, and, when S2 is finally completed, P2 will wait forever. This very mistake is often the cause of deadlocks in systems said to be "basically correct, but susceptible to heavy load." In fact, the "heavy load" is precisely what causes the execution time T(S2) to exceed the unrecognized, implicit limit T(S1).

Perhaps it might be wiser to define the emission of a signal that is not awaited as an error instead of an empty operation, i.e. to declare *awaited(s)* as a necessary precondition of *send(s)*. The culprit of system crashes might then be more easily located.

Language and System Requirements for Real-Time Programming

It is futile to make an attempt to provide an exhaustive list of necessary features and detailed language facilities for real-time programming. Instead, it seems sensible to concentrate on basic issues. It is unquestionably the form and *structure* of a language that plays the dominant role in the development of well-structured and reliable programs, be they sequential or "real-time." In this respect, a "real-time language" appears to require no more structural concepts than a good multiprogramming language. The most important single item is a notational unit for describing *processes* that are themselves purely sequential, but can be executed concurrently. In particular, this implies the absence of the notion of an interrupt. We must be able to think of each logical process as being sequential and coherent. The second item that appears to be necessary is a collection of shared variables together with their operators for which mutual interference is excluded (monitors, *interface modules*). The third item is an object to trigger continuation after waiting (*signals*).

Whereas real-time programming calls for no additional language structures, and for hardly any added facilities, a new requirement must be imposed on implementations: they must be able to provide accurate execution time bounds for any compiled statement or statement sequence. Such a facility, although unusual at present, is readily provided by additions to existing compilers. In the case of straightforward code generation without optimization, a simple table indicating the times for each kind of statement may even be satisfactory. Particularly important figures in this table are the times for the *send* and *wait* operations. They may be taken as reliable indicators for the effectiveness (or overhead) of a real-time language implementation.

The postulate that the programmer must be able to think of each logical process as being coherent, i.e. behaving as if executed by its private processor, does not of course imply that actual implementations cannot deviate from this schema as long as they satisfy the defining axioms. In reality, most implementations *share* processors among processes, and several processors may take turns in the execution of the same process. The reason for insisting that a program should not rely on assumptions about any particular processor sharing strategy is twofold: to *simplify the logical foundations* of the program and to provide implementations with a maximum degree of *freedom for choosing such a strategy*. This reason is the sole motivation behind the distinction between logical, coherent processes and physical, interruptible processors. The "ban" on the notion of interrupt is merely a corollary because interrupts pertain to shared processors, but not to coherent processes.

The acceptance of processor sharing implementations has, however, one grave consequence: the determination of execution-time bounds can no longer be based on static timing figures only. We must take into account that during execution of any statement the processor may be diverted to work on another process for an unknown "time slice." This difficulty is perhaps the most convincing reason why hitherto real-time programming remained the unchallenged domain of assembly coding, where programmers feel most confident to have all details under their control. However, if the influence of processor sharing (i.e. of interrupts) can be ignored in considerations about a system's computational state (i.e. in logical assertions) and can be confined to timing considerations only, then the goal of a discipline of real-time programming appears noticeably more realistic.

Real-Time Programming in High-Level Languages Using Processor Sharing

It does not appear feasible at this time to postulate any generally valid and at the same time practically useful rules for the determination of execution time bounds for systems using processor sharing. Rather we shall investigate techniques used to cope with real-time problems when using assembly code. We may then be able to identify a frequently used program schema that

seems manageable and to suggest a formulation in terms of high-level languages.

First and most important, we find that time-critical operations most frequently occur in close connection with peripheral devices such as readers, sensors, and equipment to be controlled. It is customary to associate a process with each device (or array of identical devices). If a separate processor is available for this purpose, the process is called a *driver*, if a general processor has to be shared, it is called an *interrupt handler*. In either case, both the device itself and the associated routine effectively constitute a pair of cooperating sequential processes. This typical situation is abstracted by the following schema, expressed for the case of an input device. The completion signal is customarily called *the interrupt*.

```
var InterfaceBuffer: Register;
      initiation, completion: signal;
process InputDevice;
begin
  loop wait(initiation); ReadData; PutData; send(completion)
  end
end InputDevice;
process Driver;
begin
  loop send(initiation); wait(completion); GetData; ProcessData
  end
end Driver
```

The characteristic of this schema is that the two processes effectively *alternate* in being active. Moreover, the first process represents an essentially nonprogrammable device. In Modula, this typical example of cooperating sequential processes is therefore condensed into a single so-called *device process*. The *send*(*initiation*) statement is replaced by commands determined by the specific hardware device, and the *wait*(*completion*) statement is replaced by a generic statement called *doio*, representing the device's entire activity.

```
process Driver;
  ⟨declarations of device status and buffer registers⟩
begin ⟨initializations⟩;
  loop StartDevice; doio; GetData; ProcessData
  end
end Driver
```

The driver communicates with other processes; in the example this is hidden in the statement *ProcessData*. Communicating procedures would have to be declared in an interface module. Since, however, the statements in the cycle (apart from *doio*) are few and take a very short time anyway, it appears appropriate to move the entire driver process text into that interface module and to explicitly declare the *doio* statement to be exempted from the mutual exclusion condition. This kind of special interface module is called *device module* in Modula. The *doio* statement and declarations of hardware specific facilities, such as device control and buffer registers, are allowed in such device modules only.

```
device module Reader;
  define read;
  var buffer: BufferType;
    nonfull, nonempty: signal;
  procedure read(var x: character);
  begin if . . . then wait(nonempty) end;
    fetch(x); send(nonfull)
  end read;
  process Driver;
    var CardReaderBuffer: Register;
        CardReaderStatus: Register;
  begin initialize;
    loop StartReader; doio;
      if . . . then wait(nonful) end;
      get character from CardReaderBuffer and deposit;
      send(nonempty)
    end
  end Driver;
begin initialize local data and buffer;
  Driver
end Reader
```

We now return to the important question of how to take real-time constraints into consideration. Remember that we cannot rely on timing information obtained from the static program text since we assume that the processor may be shared among the various processes. Evidently we need to know something about the strategy by which it serves individual processes and, in particular, whether or not certain processes are served with priority.

The common technique is that so-called interrupt routines are given priority, i.e. that upon receipt of an interrupt signal the processor is unconditionally diverted to that routine. The analogy now becomes apparent: because the device process corresponds to the interrupt routine, it must be executed with priority in noninterruptible mode. In systems with several devices, interrupts are queued when arriving while another is being served. In Modula, this effect therefore manifests itself as a hidden delay statement behind each *doio*. Its minimal duration is 0, and the maximum duration d is equal to the sum of the execution times of all other interrupt responses: $0 \leq d \leq \text{sum } T(S_i)$, where $T(S_i)$ is the time to execute the longest statement sequence S_i between any two instances of *doio* in the ith device process, measured in noninterrupted mode, and where i ranges over all other processes.

The recommended discipline of real-time programming in Modula can now be summarized as follows:

1. Confine time-dependent program parts to device processes which are executed in noninterruptible mode.
2. Execution time of statements in device processes is determined statically, i.e. as if there were no processor sharing.
3. Each *doio* statement is assumed to be followed by a hidden delay statement whose bounds are given above.

Note: The lower bound of the delay is not exactly 0, but is equal to the "overhead" of processor switching upon interrupt. This is another characteristic perform-

ance figure of a multiprogramming language implementation.

In practice, the typical system operates a large number of devices, a few of which require fast response. As no appreciable delays in the associated drivers are tolerable, the value d – being the sum of a large number of terms – might easily become unacceptably high. The solution lies in the use of a priority interrupt system. Its essence is that the shared processor, when operating at priority level p, is interruptible only by devices assigned an interrupt strength $k > p$ ($0 \leq p \leq pmax$, $0 < k_i \leq pmax$). Processors without this feature appear as a special case with $pmax = 1$.

Unfortunately the situation becomes hopelessly more complicated and eludes exact analysis because now device processes can be interrupted by other device processes of higher priority. Yet, the situation becomes manageable again if we adhere to the following constraints:

1. Every device process P_i is cyclic, the cycle consisting of a statement sequence S_i executed by the shared processor and the statement *doio*, representing the waiting for the device completion.
2. The cycle time

   $t_i = T(S_i) + T(doio_i)$

   of process i at any priority level k is considerably larger than that of all processes at level $k - 1$. Note that $T(doio_i)$ is determined by the ith device.
3. Over every cycle, the ratio

   $r_i = T(S_i)/(T(S_i) + T(doio_i)) \ll 1$.

   This is the fraction of time that the processor is needed to serve the ith device process.

Under these conditions we may approximately compute the effective execution time T of any statement S in a device process at priority level k as

$$T(S) = T'(S)/(1 - \text{sum } r_i),$$

where $T'(S)$ is the execution time of S without processor sharing, and where i ranges over all processes at priority levels greater than k.

Let us now formulate, as a brief example, a time-dependent card reader driver in terms of Modula:

```
device module CardReader [priority];
  define read;
  var buffer: BufferType;
  procedure read( ); . . .
  process Driver [interrupt address];
    ⟨declarations of device registers⟩
  begin
    loop . . . StartMotion;
      loop doio (*delay may occur here*);
        when EndofCard exit
        fetch next character from device register;
        deposit in buffer
      end;
      deposit EndofCard mark in buffer
    end
  end Driver;
begin initialize local data and buffer;
  Driver
end CardReader
```

Assuming that reading a character takes t = T(*doio*) milliseconds, we must conclude that the necessary constraint is

T("{(*delay*)}; **when** . . . deposit") < t.

If no other device drivers operate at priority level greater than or equal to p, the hidden delay is zero. If no other driver operates at a level greater than p, the execution times can be determined statically without considering a slowdown factor.

Problematic Priority

We have seen that the notion of process priority is necessary if real-time programs are to be executed by systems using processor sharing. Even so, explanations of a program's validity are often intricate and complex. Unfortunately the notion of priorities also imports a dilemma in connection with the presented concept of synchronization through signals.

We started out with the recommendation to design programs in a timing-constraint-free manner whenever possible and instead to rely on logical assertions on the state of the computation. The signal facility must therefore also be defined in an axiomatic way. The following is commonly accepted:

{P_s & Q} send(s) {Q},
{Q} wait(s) {P_s & Q}.

Here Q is a condition over the variables of the interface module in which the signal s is declared. Q must hold before and after each *send* and *wait* (and *doio*) statement, and also at the entrance and exit of each interface procedure. Q is called the *interface invariant*. P_s is the condition associated with the signal s. It must be established before each signal emission and is therefore guaranteed to hold whenever a process resumes after waiting for the signal. This definition of the semantics of signals – including the association of a transmitted condition (message) – is certainly sensible and intuitively correct. However, it also has the effect of enforcing an effective switching of the processor for each *send* statement (when sharing is employed), for, if a process would not immediately be resumed after signal receipt, no guarantee could be given for the condition P_s still to hold when at a later time the waiting process obtains the processor. After all, other processes might have invalidated P_s in the meantime [5].

But now the following awkward question arises: Should the processor still be switched if the sending process is declared to have a higher priority than the receiving one? If the answer is yes, unreasonably frequent processor switching may be the consequence, and – worse – timing considerations become exceed-

ingly difficult if not impossible. If the answer is no, then the axioms governing signals must be revised, and they become inherently more complex. Considering that signals are a truly basic concept, that would look like a symptom of inadequacy.

The question remains open. In Modula, the negative answer was adopted. It seems counterintuitive to release a processor to a process of lower priority. The following additional constraints on signals emitted by device processes are sufficient, but perhaps too strong, and certainly cumbersome:

1. Each signal *s* which is emitted by a device must be awaited by a single (regular) process only.
2. A device process must never itself invalidate the condition associated with a signal which it emitted.

Acceptance of these rules yields the benefit of applicability of the presented framework of reasoning about timing constraints and the possibility of an extremely efficient implementation of signalling. The latter is truly important for real-time applications and low-level device handling.

Summary and Conclusions

We have divided programming into three categories with ascending complexity in required reasoning: sequential programming, multiprogramming, and real-time programming. The former two categories are characterized by the property that the validity of programs is independent of the execution speed of processors used. In order to keep real-time programs intellectually manageable, we recommend that they first be designed as time-independent multiprograms and that only after analytic validation they be modified in isolated places, where the consequences of reliance on execution time constraints are simple to comprehend and document.

The advantages of high-level languages are even greater in real-time programming than in sequential programming. This is because appropriate language structures help in isolating those parts of a program that rely on timing conditions (device processes) from those parts where time is irrelevant. The important idea is that all processes can be considered as purely sequential and coherent. In many cases, the consequences of using an actual system with a shared processor are relatively simple to take into account: they manifest themselves as a virtual delay after each *doio* and, if the implementation is based on a priority interrupt system, by a virtual slowdown of actual processor speed by a computable factor.

The notion of priority is important when dealing with systems using the technique of processor sharing. The presented rules apply only if the processes of any given priority level require an order of magnitude less processor time than all processes at the next lower level. In all other cases, the problem of analytically verifying a system's validity appears formidable, if not impossible. Then, as often in engineering, it is wiser to avoid the problem than to solve it. With rapidly decreasing hardware cost, it is best to avoid the technique of sharing and to dedicate a processor to each process. This appears even more attractive if we consider problems of reliability in the sense of hardware fault tolerance. This subject was not covered here, as it is based on engineering experience rather than language design.

In summary, the use of a suitable high-level language, together with adherence to a strict programming discipline, is instrumental in making analytic validation of real-time programs possible. It may be the only viable way towards genuine reliability in real-time systems.

Acknowledgments. The author gratefully acknowledges the valuable comments made by S.L. Graham, S. Knudsen, J.H. Morris, and I.C. Pyle, and the support received from the Xerox Corporation.

References

1. Brinch Hansen, P. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, N.J., 1973.
2. Dijkstra, E.W. Cooperating sequential processes. In *Programming Languages*, F. Genuys, Ed., Academic Press, New York, 1968.
3. Hoare, C.A.R. Monitors: An operating system structuring concept. *Comm. ACM 17*, 10 (Oct. 1974), 549-557.
4. Howard, J.H. Proving monitors. *Comm. ACM 19*, 5 (May 1976), 273-279.
5. Howard, J.H. Signaling in monitors. Proc. 2nd Int. Conf. on Software Engineering, IEEE Cat. No. 76CH1125-4C, San Francisco, Oct. 1976, 47-52.
6. Wirth, N. Modula: A language for modular multiprogramming. *Software—Practice and Experience*, 7, 1 (Jan. 1977), 3-35.