FORMAL SPECIFICATIONS FOR
REAL-TIME SYSTEMS

Richard M. Cohen

October 1978

ICSCA-CMP-14

CERTIFIABLE MINICOMPUTER PROJECT
Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, Texas  78712
ABSTRACT

Specifying real-time constraints for programs is much more complex than merely specifying partial correctness. This results in part from the need for a much more complete description of the program's execution environment. Some of these trouble spots for specifications are discussed. An extension to the specification facilities in Gypsy to handle certain sorts of timing constraints is proposed.
## TABLE OF CONTENTS

1.0 WHAT IS REAL-TIME PROGRAMMING? .................................. 1

2.0 WHAT IS PERFORMANCE? .................................................. 2
   2.1 Time Dependence Versus Time Independence ....................... 3

3.0 SPECIFICATION OF PERFORMANCE REQUIREMENTS .................. 3

4.0 ANALYSIS OF PROGRAM PERFORMANCE ................................ 4
   4.1 Concurrency And Multiprogramming ............................... 6
   4.2 Empirical Analysis .................................................. 6

5.0 LANGUAGE REQUIREMENTS FOR REAL-TIME PROGRAMMING ........... 7
   5.1 Sequential Aspects ................................................. 7
   5.2 Error Recovery And Handling ..................................... 8
   5.3 Predictable Time And Space Usage ............................... 8
   5.4 Concurrency And Real-Time Programming ........................ 9
   5.5 Interrupts As Service Requests .................................. 10
   5.6 Dynamic Task Creation Not Required ............................ 11
   5.7 Scheduling .......................................................... 11

6.0 REAL-TIME PROGRAMMING SYSTEM SPECIFICATION CAPABILITIES .. 12

7.0 A PROPOSED SPECIFICATION METHOD ............................... 13

8.0 SUMMARY .............................................................. 17

REFERENCES ............................................................... 18
1.0 WHAT IS REAL-TIME PROGRAMMING?

Real-time programs are programs that have real-time constraints. Wirth has suggested the term "processor time dependent" programming. [Wirth, 77] This does not distinguish a small portion of existing programs. In fact, all programs have some real-time constraints! Most programmers are not willing to wait days for job turnaround. When one submits a job to the computer, there are implicit real-time performance constraints. The constraints are usually quite easily satisfied by the system, and are not tight bounds. (He doesn't care whether turnaround is 5 minutes or 6 minutes, but 24 hours is unacceptable.) The real-time constraints are often on the "system" as a whole, rather than on a particular program/task within the system. These constraints are most often very loose, and unstated. They are very easy to attain, and are not critical. They are usually not termed "real-time constraints" at all, but are called "system performance." They are constraints on the combined set of processes being handled by the computing system. It is only when the real-time constraints become more precise, more critical, and, perhaps, harder to meet that we enter the acknowledged area of "real-time programming."

Real-time programs are characterized by their interactions with external devices, and by timing constraints relating some of these interactions. Typically these constraints require the program to keep pace with an external, on-going event, which is running concurrently with the monitoring or controlling process. It is often the case that a single computer system will be monitoring a set of concurrently proceeding external processes (e.g., data collection during an experiment, process control -- possibly with feedback -- driving a peripheral device such as a card reader.) Interestingly, many real-time applications do not place absolute real-time constraints. A disk driver may be allowed to miss a revolution occasionally. A telephone switching system may lose a call occasionally, but the vast majority of the calls must be handled promptly and properly. This sort of
Formal Specifications for Real-Time Systems

specification implies that occasionally the program may not execute fast enough, but most of the time it will. The program may or may not have to recognize this failure, and alter its behavior accordingly.

Also, real-time programs and real-time programming systems may not produce "answers" when run (as contrasted with applications programs, which do produce identifiable "answers.") [Sammet, 71]

Real-time constraints on program performance stem from program interaction with external, asynchronous processes. The process may be some on-going event, a peripheral device, or even the programmer himself. The constraints arise because the external and internal events must at some point be synchronized, and the external process has some expectations / requirements regarding when that synchronization will take place. A peripheral device, such as a disk drive, may be viewed as an external process, running in parallel with the user's program, which accepts certain requests, and provides certain responses. But there are timing constraints on when the device will accept requests, and on when the program must accept the responses from the device.

2.0 WHAT IS PERFORMANCE?

Program performance is a very vague term. It has been applied to almost any execution time characteristic of a program. And rightly so, for they are all performance characteristics. We may be interested in the execution time of a program, or the space used, or how the space is used, or some sequence of internal events within the program. We can view many of these properties from either a time independent perspective, or from a time dependent one. Even within the time dependent viewpoint, we may be concerned with "real time," "virtual time," or simply with execution sequences. "Real time" measures events with respect to a standard time frame external to the programming system. "Virtual time" measures events within a time scale of processor service time accumulated by the program. Execution paths are concerned only with relative order of events looking at the sequence of events without regard to specific event times.
Formal Specifications for Real-Time Systems

2.1 Time Dependence Versus Time Independence

The characterization of "real-time performance" conjures the image of a trace of external program events measured along program execution time. This is reasonable, whether the events are page faults, or disk accesses. However, the analysis of real-time properties may depend on the analysis of other time dependent (but real-time independent) performance. For example, the execution time of a program depends on its flow of control, but the analysis of the flow of control is independent of the program's execution speed. The analysis of flow of control may be done statically or dynamically. The dynamic, empirical, results may relate to actual execution times, or to frequency counts within the program. (Corresponding to consideration of real time or virtual time.)

Multiprograms (programs containing several internal concurrent tasks) must use protocols for synchronization and coordination. We may wish to analyze the program protocols empirically, by tracing relevant events in virtual time or real time, or deductively by examination of the flow of control within the program. Usually the protocols are independent of particular hardware or software systems. (The protocol may only be relevant to a particular system, but the program behavior is independent of that.) Analysis of program paging behavior is very interesting, in that it depends not on the high level language program, but rather on the machine code program generated by the compiler. Certainly the machine code depends on the source program, but it is clear that paging behavior depends on a particular compiler/virtual memory hardware/software system.

3.0 SPECIFICATION OF PERFORMANCE REQUIREMENTS

Real-time constraints arise from program interaction with external processes. This suggests that any real-time specifications for the program must have their roots in the properties of those external processes. It also suggests that the natural point to give those specifications is at the interface points. One of the unfortunate consequences of this is that these specifications may filter down through the program design and decomposition in very non-obvious ways. Knuth's "Empirical Study of Fortran Programs" reported the 90-10 phenomenon that most programs spend 90 per cent of their execution time in 10 per cent of their code. However, exactly
Formal Specifications for Real-Time Systems

which 10 per cent of the program code that is frequently executed is often not obvious. (Little portions of programs, such as lexical analyzers, tend to take up much more time than one might guess at first.) Without analysis of the actual algorithm and program code, it may not be clear where performance constraints can or cannot be met within the design decomposition. Examples from Wirth [Wirth, 77], and Phillips and Bredt [Phillips, 76] suggest that only the high level specifications are clear. Lower level specifications are not derived during decomposition. They are derived by relating the decomposition and the actual performance (or predicted performance) for internal processes and modules. This corresponds to a bottom-up approach to specifications.

One of the reasons that performance specification is so hard is due to the (common) presence of concurrency. In describing a set of concurrent processes, you may be able to state performance specifications for single processes, or for several processes together. However, typically, the behavior of one process will depend on that of one or more of the others. When a single program is decomposed into several subprocesses, it is not clear how to decompose the performance specifications for the top level program into realistic specifications for the individual subprocesses.

4.0 ANALYSIS OF PROGRAM PERFORMANCE

Analysis of program performance seems to occur at several levels. These levels correspond to progressive commitments to particular implementation (both in software and in hardware.)

1. Computational complexity of the problem. At this level the analysis deals only with the problem itself, and treats properties of all possible solutions. The analysis at this level may be very abstract.

2. (Abstract) analysis of algorithms. This binds a particular algorithm to the given problem. This is execution on an abstract machine (e.g., for sorts the abstract machine executes only comparisons and moves.) This analysis may be more concrete than level 1, but is still quite abstract compared to real-time performance. (It may be necessary, however, for the predictive analysis of real-time performance.)
Formal Specifications for Real-Time Systems

3. **Analysis of (concrete) programs.** This binds the program to a particular programming language, and the algorithm to a particular implementation in that programming language. This level is susceptible to empirical testing. The FORTUNE system of Ingalls [Ingalls, 72] provides execution time profiles for FORTRAN programs, by running an instrumented version of the program. The validity of the profiles depends on the user's choice of test data, and whether that test data represents the data normally encountered during program execution. Also, we must be sure that the instrumentation code does not interfere with normal execution of the program code. Wegbreit's METRIC system [Wegbreit, 75] attempts to perform deductive analysis on simple LISP programs to predict various performance characteristics. Complete deductive analysis at this level may not always be possible, because compiler/optimizer code generation, which may affect actual program performance, must be taken into account.

4. **Analysis of Compiled Code.** This level may involve examining the output of the compiler/code generator, or it may be an extension of concrete program analysis to include knowledge of the particular code generator to be used. This corresponds to binding the program to a particular compiler/code generator. (This may not bind the compiled code to a particular hardware configuration, as the same code can often run on several different configurations or CPUs from a "family" of computers.)

5. **Analysis of execution behavior on a particular hardware configuration.** This level finally binds the hardware processor and external devices. The external devices may appear in the analysis at higher levels, but usually need not be bound to particular equipment characteristics until this level. This level of analysis is particularly susceptible to empirical testing. Hardware monitors, and software monitors using parallel processors (such as the SPY system for the CDC 6000 [Jasik, 72]), are quite common. They can only provide statistical information on program behavior. One interesting system for software measurement is the Informer system of Deutsch and Grant [Deutsch, 71]. It attempts to deduce some characteristics of the run-time performance of software measurement routines that are added to the operating system dynamically for measuring user program performance.
Formal Specifications for Real-Time Systems

4.1 Concurrency And Multiprogramming

This level of analysis is somewhat different from the progression established by the others. The basic problem is to take a set of sequential programs, each of which may have been analyzed individually, and analyze their behavior operating as a set of concurrent processes under a particular scheduler. (This may be thought of as "binding the scheduler.") This will require knowledge of the scheduling policy, and specifications. Task performance may be affected by other tasks' dynamic requests for resources (including the CPU.)

The set of tasks may be either dependent or independent (or subsets of tasks may be one or the other). If the tasks interact, then we must worry about their synchronization protocols. We must consider the correct implementation of the protocol within a single task, as well as the correctness of the task interactions that result. The scheduler policy must be analyzed concerning possibilities of deadlock, and whether resources will be allocated "fairly." (That is, they will be allocated in such a way that all tasks have reasonable access to the resources they need.) (Analysis of operating system performance typically is done at this level, with the operating system running concurrently with several independent user programs.)

Analysis of concurrency cuts across the other "levels" given above, and may be done at the abstract or concrete level. This is because concurrency may appear at any level in the design and development of an algorithm or program. (It may also be introduced at any level. This includes the compilation process, as compilers for machines such as the TI ASC will introduce parallelism into compiled code where none existed explicitly in the source code.)

4.2 Empirical Analysis

The most common form of empirical program testing is program debugging. Correctness of results is one important aspect of run-time behavior. Testing is also used for performance measurement, whether of a single program, or a large, multiprogrammed operating system. The performance data may be used to "tune" or enhance system performance. (In the realm of real-time programs this may be considered debugging!) The
Formal Specifications for Real-Time Systems

performance data may relate to execution speed, job throughput, page fault rate, or many other possible measures.

In a dedicated hardware environment the measurement may be made by observed external behavior (such as watching a wall clock) or by use of hardware monitors [e.g., Jasik, Batson, Graham,...]. In a software environment, various conditions and monitoring must be done by simulation and software probes. For example, in a non-dedicated system, hardware interrupts are masked by the operating system, but may be simulated for the user task [Barnes, 76]. Similarly, interrupts generated by the user task may be trapped by software probes [Deutsch, 71] or by a monitoring process [Thomas, 75].

Ingalls' FORTUNE system [Ingalls, 72] instruments a FORTRAN program to produce an execution profile. The profile indicates how many times each statement in the program was executed. FORTUNE also provides a simple weight for each statement. The weight is an estimate of the relative execution time for the statement, based on lexical analysis of the statement. (FORTUNE was originally called FETE - FORTRAN execution time estimator.)

Cohen and Carpenter [Cohen, 77] take a "shotgun" approach. Virtually every execution time event (at the source language level) is recorded. Then, after execution, this large database can be queried concerning various events during execution.

5.0 LANGUAGE REQUIREMENTS FOR REAL-TIME PROGRAMMING

5.1 Sequential Aspects

The sequential portion of our programming language should offer all of the popular (much debated) desirable features to support "good" programming. The list includes modularity, data abstraction, control structures, and so on. Efficient implementation for data structures (such as queues and tables) may be especially important. Brinch Hansen has stated that only about 4 per cent of the code for the SOLO operating system required the concurrency features of Concurrent Pascal. (About 96 per cent of the SOLO system was written in Pascal/Concurrent Pascal.) [Brinch Hansen, 77, page 78]
Formal Specifications for Real-Time Systems

Real-time programs do require more error handling capabilities than most sequential programs. Batch programs often execute in virtually fault-free software environments. In the event of a fault, it may be acceptable to abort execution and let the programmer apply some appropriate correction and resubmit the program run. In a real-time situation, there is often a (fairly high) cost associated with (excessive) delay or failure. (Shelley and Mischelevich [Saffer, 75] have even suggested this as one of the defining characteristics of a real-time program.)

5.2 Error Recovery And Handling

"Graceful degradation" is a desirable attribute of a real-time system. In the event of some external fault preventing the system from performing its full task, the system should continue to perform some portion of its task (perhaps at a reduced service rate). The system should not simply halt operation. An error recovery mechanism in the language may be necessary to respond to some sorts of faults. Some checks at the problem domain level can be programmed explicitly, of course, and these put no special error handling requirements on the programming language.

5.3 Predictable Time And Space Usage

We must be able to predict the elapsed real time during execution of a routine. Thus we must be able to analyze, at least statistically, the sequence of machine instructions to be executed with respect to running time. Some software mechanisms do not admit this sort of analysis. For example, LISP provides a virtual machine in which explicit free space management can be forgotten by the programmer. The garbage collector is invoked as a co-routine by the LISP program at unpredictable intervals. Thus the real-time execution of the LISP program is delayed by however long the garbage collection takes. (This argument has been used to justify discussion of a hardware garbage collector to run in parallel with the LISP processor [Steele, 75]. Parallel garbage collection may not be good enough if the list processor can still be delayed unpredictably waiting for the garbage collection.)
We should also be able to predict the program's space requirements. It may very well be catastrophic to run out of space at execution time. (Of course, our error handling facility will catch the error, but it may not be possible for the program to continue without the needed space.) Space requirements may include primary and secondary storage. While exact storage requirements may not be needed, some degree of predictability is needed during the program design. Predicting exact space usage at run time may involve considerable analysis of the complexity of the program (e.g., stack usage for a recursive routine). One problem cited with the RT1 system [Barnes, 76] was the unpredictability of the run-time stack size. Stack size is easily predicted in the absence of recursive routines. (RT1 allows both recursion and dynamic creation of processes.) The Modula compiler computes stack size if there are no recursive routines. If recursive procedures are used, then the user must supply stack size requirements via compiler directives.

5.4 Concurrency And Real-Time Programming

Concurrent programming arises quite naturally in real-time programming. Programs are executed on a processor that is connected to some external devices (typically various input/output equipment, perhaps interfacing the processor with a mechanism to monitor and control an ongoing process such as a manufacturing process or the trajectory of a missile). These external devices operate asynchronously with respect to the main frame processor. In fact, they almost never run off the same "system clock" as the main processor. Even if they did, the parallelism involved would lead to the notion of concurrency in program description, even though, in this case, the tasks would be synchronous rather than asynchronous. Even with synchronous tasks, it is often easier to describe the tasks separately rather than as a single sequential system. Discrete simulations fall into this category.)

External devices run concurrently with the central processor and synchronize with it according to appropriate hardware interface protocols. In general, these hardware protocols have peculiarities not normally present in discipline software protocols (e.g., timing constraints, "ready-or-not-here-I-come" actions). It is natural to isolate interactions with each of these external processes to a single internal process. Then all program interactions interface with this internal process
according to disciplined software coordination protocols.

Real-time programs often monitor and control several external processes, or several distinct aspects of a single external process. This also leads to decomposition into a (functionally) disjoint set of tasks. This disjoint set of tasks may be expressed as operating in parallel. Whether such a description fits will depend on the particular situation. If the tasks have different real-time scheduling or response characteristics, their description as distinct concurrent processes may be simpler than their description as a single integrated sequential task.

5.5 Interrupts As Service Requests

Dijkstra and Wirth (and others) have pointed out the problems associated with the notion of hardware interrupts. Interrupts are used to simulate parallel processing and to implement scheduling for real-time constraints. Wirth has proposed that we replace this simulated parallelism with explicit parallel processing [Wirth, 69]. Languages such as Modula and Concurrent Pascal support parallelism and real-time programming. The hardware interrupts are buried in the virtual machine provided by the programming language. These languages leave scheduling of the parallel processes to the hardware demands of the virtual machine (possibly realized by interrupts in the hardware.) (PDP/11 machine dependent) Modula provides for directly coupling processes to the hardware interrupt vector. The interrupt signal can be considered a dynamic scheduler request for a context switch to the interrupt handling process. Software synchronization of processes must be handled explicitly by the program. Concurrent Pascal provides for coordination by use of monitors. Modula provides synchronization by use of monitors (called interface modules), too.

HAL/S [Intermetrics, 74] goes considerably farther in allowing processes to be scheduled for activation in an extremely flexible manner. The virtual machine provided by HAL/S includes a priority scheduler to schedule active tasks. HAL/S allows dynamic task creation and explicit scheduling requests (e.g., requests for regular task activation, dynamic changes in priority, etc.).
5.6 Dynamic Task Creation Not Required

We have seen two sources of concurrent tasks in real-time programming. In both cases, the concurrent tasks arise as a result of static properties of the program environment. Thus the task decomposition is known at compile time. This suggests that perhaps dynamic creation of tasks at run time may not be necessary. Both Modula and Concurrent Pascal have followed this path and assume that all tasks will be created at system initialization time. HAL/S does provide for dynamic creation of tasks, but only one activation of a task block is allowed at a time! (This difference is partly explained by the fact that Modula and Concurrent Pascal are intended to be appropriate for constructing stand-alone system software, whereas HAL/S assumes somewhat more elaborate run-time support.)

5.7 Scheduling

We have already seen that real-time programming and concurrent processing are closely related. Languages supporting concurrency must have some statement of how the parallel processes are executed. The virtual machine model introduced by the language definition usually specifies that each parallel process executes on its own virtual processor. This is to be expected, as an informal (or formal) semantic description of the language is not concerned with details of a specific implementation on a specific hardware configuration. This is also due to the fact that most semantic definitions (and other language definitions) ignore execution timing considerations. However, in discussing real-time programming, we must be very concerned with just these sorts of details. For example, when only a single processor is provided, all processes must share it. Language level coordination primitives can be viewed as allocation (or scheduler) requests for access to the processor. Thus, we quickly arrive at the notion of an explicit resource scheduler. While this notion may be left quite vague in most language definitions, our analysis requirements dictate a need for more specific models of scheduler behavior. It is not yet clear exactly what scheduler characteristics are needed for execution time analysis.

Scheduler policy is certainly important. For example, a "pie-slice" scheduler that allocates a fixed slice to a process (and all of its descendants) would permit a degree of modularity
Formal Specifications for Real-Time Systems

in a timing proof, since the processor service for a given process would not be affected by competition from its siblings. However, a "pie-slice" scheduler might be inappropriate for some real-time applications. A priority driven scheduler is another possibility. In fact, a fixed priority, interrupt driven scheduler is quite appropriate for a number of real-time applications. A brief treatment of this scheduling policy in the context of the language Modula has been done by Wirth [Wirth, 77].

The scheduler can be analyzed for fairness and potential deadlock. (Certainly the user program must also be analyzed to assure proper use of higher level protocols to avoid deadlock, and to assure fairness, where appropriate.) More investigation is needed to determine the relation between real-time analysis and various properties of schedulers. Many scheduling problems can be modelled and analyzed as Markov processes. This, too, should be investigated.

6.0 REAL-TIME PROGRAMMING SYSTEM SPECIFICATION CAPABILITIES

The modern concept of programming requires the generation of formal specifications for a program as well as executable source code [Good 77, Liskov 75]. The specifications are an integral part of design and explicitly support performance predictions, formal proof of program properties, definition of testing requirements and execution time validation of consistency between specifications and code. The most effective use of specifications in real-time programming will, in the near future, be in system design analysis and in testing and execution time validations. Many dependencies of timing relationships are not visible in static specifications or source code but depend on future bindings to hardware systems. It is possible, nonetheless, to define from the required properties of programming language requirements, some characteristic of the specification language for real-time programming.

The following general properties must be specific:

The external events for which responses must be generated must be characteristic with respect to their response requirements and with respect to their occurrence patterns.
The resource requirements for responding to an event must be specific. These specifications may include processing requirements and memory requirements.

Execution environment constraints such as aggregate resource requirements or time constraints for a set of events may need to be specified.

There has been little serious attack on the problem of specification for real-time programming. The next section describes an example of how real-time programming specifications might be added to an existing static proof-oriented specification language.

7.0 A PROPOSED SPECIFICATION METHOD

The programming language Gypsy [Good, 78] has been developed to support the formal specification of system software. We propose a method of specifying some aspects of real-time programming as an addition to Gypsy specification capabilities. The example demonstrates how specification capabilities can be used to verify that a time constraint was indeed met.

Gypsy allows parallel processing with concurrent processes synchronizing via message buffers [Brinch Hansen, 73]. Concurrent processes cannot share data objects directly; all communication and sharing is done via the message buffers. By recording transaction histories of the messages sent to a buffer by each process and the messages received from a buffer by each process we can completely capture a process' interaction with its (hardware and software) environment. The sequence of messages received by process "P" from buffer "B" is denoted:

\[ \text{infrom} (B,P) \]

Similarly, the output history for process "P" is denoted:

\[ \text{outto} (B,P) \]

This history notation has been extended to pair each message with a time stamp indicating when the message was sent or received. (Input histories are stamped with RECEIVE times; output histories are stamped with SEND times.) The only axiom concerning these time stamps is that within each history the times monotonically increase. (This follows from the fact that
messages are received in the same order that they were sent.) These extended histories are denoted "xinfrom(B,P)" and "xoutto(B,P)". The function "time" maps an element of an extended history sequence into a time stamp. Gipsy allows elements of a sequence to be selected by using subscripts. The elements are always indexed from 1 to the size of the sequence. Thus, "time(xinfrom(B,P)[i])" denotes the time stamp of the i-th message received by process "P" from buffer "B".

The "infrom" and "outto" histories are called local histories because they represent the history of message traffic controlled by a single process or procedure (including any procedures it invokes). The global histories of a buffer includes all messages sent or received by any process. The input and output global histories are denoted "allfrom(B)" and "allto(B)", respectively. The extended (timestamped) global histories are denoted "xallfrom(B)" and "xallto(B)".

We can now begin to specify real-time constraints. We can use the difference between various time stamps to describe the real-time difference between the program send and receive actions that caused the corresponding message transactions. For example,

\[
\text{time(xoutto(responses,P)[i])} - \text{time(xinfrom(requests,P)[i])} \leq \text{MaxResponseTime}
\]

specifies that each request received by process "P" must be followed by a response within "MaxResponseTime". This specification expresses a constraint on process "P". Specification of systems of concurrent processes become more complex.

Let us examine a slightly more complex example. The system comprises three processes.
Formal Specifications for Real-Time Systems

```
<table>
<thead>
<tr>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Responses</td>
</tr>
<tr>
<td>Requests</td>
</tr>
</tbody>
</table>
|------|--|--
| Disk | -->Commands--> | Disk |
|      | <---Status<--- | Sub- |
| Driver| <---Data<----- | System |
```

There is a user process making read-only requests for data from a disk file. The user process sends requests to the disk driver, and receives the requested data in response. The driver communicates with the disk system (e.g. a disk controller and a disk drive) to obtain the requested data. We will look at the specifications of the disk driver process, based on the real-time constraints of the disk system itself.

It is assumed that to obtain the desired data from disk, the driver must:

1. Issue a read request,
2. Wait for a Ready/NotReady response from the disk system,
3. Reissue the read request if the access failed,
4. Accept the data from the disk within "MaxResponseTime" after the Ready response from the disk system.

A Gipsy procedure for process DiskDriver is outlined below. In Gipsy all compound statements (e.g. Loop, If...then) end with the keyword "end". The assertion in the inner loop gives the basic timing constraint. The program loops sending the disk command until it receives a "Ready" status, and then reads the data block. The assertion states that the data transfer is initiated within "MaxResponseTime" of receipt of the ready status message.
Formal Specifications for Real-Time Systems

procedure DiskDriver(...)

  loop
    receive R from Requests;
    loop
      send R to Commands;
      receive S from Status;
      if S = Ready then leave end;
    end;  { loop retrying command }
    receive Blk from Data;
    assert
      time(last(xinfrom(Data,myid)))
      - time(last(xinfrom(Status,myid)))
      < MaxResponseTime;
    send Blk to Responses;
  end;  { loop to process next request }

The loop assertion provides the inductive basis for a procedure specification

all i:integer,
    i in [1..size(infrom(Data))]
  -> time(xinfrom(Data,DiskDriver)[i])
    - time(NthReadyMsg(Status,DiskDriver,i))
  < MaxResponseTime.

This expresses an external specification of the procedure disk driver. The specification may be imposed by its required interface with the disk process. The external specification for the disk process would be slightly different. Where the specifications for DiskDriver only describe its own interactions with the DiskSystem, the external specifications for the DiskSystem must describe all accesses by all processes. Thus we will use the global histories in this specification.

all i : integer
    i in [1..size(allto(Data))]
  -> time(xallfrom(Data)[i])
    - time(GlobalNthReadyMsg(Status,i))
  < MaxResponseTime.
Formal Specifications for Real-Time Systems

To verify this constraint requires global knowledge of all processes with access to the disk status and data buffers. This is quite reasonable, since we are expressing a constraint that must be satisfied globally by all processes together.

8.0 SUMMARY

Real-time programming is seen to be subject to the formal and disciplined specifications, proof and validation procedures now being developed and applied to programs whose behavior is well decoupled from real time of dependent external events. Some representation capabilities required in the specification and coding aspects of a programming language for real-time programming has been described. An example given as an extension to an existing programming system illustrates the potential power of formal methods in application to real-time programming.

Key issues for research are: development of technique for applying top down design to real-time problems and extending specification capabilities to stipulate (or bound) the vast uncertainties between source language programs and their execution behavior.
Formal Specifications for Real-Time Systems

REFERENCES


[Ingalls, 72] D. Ingalls. "The Execution Time Profile as a Programming Tool" in [Rustin, 72].

Formal Specifications for Real-Time Systems


