Concurrent Programming with Time

A Research Proposal

Kevin Jeffay

Department of Computer Science
University of Washington
Seattle, WA 98195

Technical Report 87-10-03
October 1987

This work was supported in part by a grant from the Washington Technology Center and a fellowship from the IBM Corporation.

Concurrent Programming with Time

A Research Proposal

Kevin Jeffay
Department of Computer Science
University of Washington
Seattle, WA 98195

1. Goals

This research is ultimately concerned with the development of verifiable real-time software. By verifiable, we are referring to the fact that one of the distinguishing features of real-time programs is the existence of a set of timing constraints which must be satisfied by an implementation of a program in order for that program to be considered correct. We propose to develop a programming system wherein real-time programs can be constructed and automatically verified to be temporally correct relative to a set of timing constraints. This programming system will consist of a concurrent programming language, compiler, run-time system, and code analyzer. Each will be described in more detail below.

2. Related Work

While there has been much work in concurrent programming languages (e.g. Mesa [Lampson & Redell 80], Ada [US DoD 83]), few, if any, have addressed the problem of verifying adherence to timing constraints. Some notable exceptions in this area are the languages Modula [Wirth 77], ESTEREL [Berry & Cosserat 85], and Real-Time Euclid [Kligerman & Stoyenko 86], and the analysis techniques of Shaw [Shaw 87].

Modula has a limited framework for reasoning about and expressing timing constraints in a program. Only processes that interface directly with physical devices are allowed to have timing constraints. In particular, this precludes the formulation of a timing constraint based on a data object. For example, one could take in data from a device in real-time but there is no elegant way to guarantee that the data can also be processed in real-time. The verification process is premised upon a static priority assignment to the device processes. Like all real-time, uniprocessor, process scheduling schemes based on static priority assignments, Modula suffers from an inability to fully utilize the processor for certain priority assignments [Liu & Layland 73]. Real-Time Euclid allows one to express more general timing constraints but must rely on an exponential time knapsack style algorithm to determine if all timing constraints can be met.

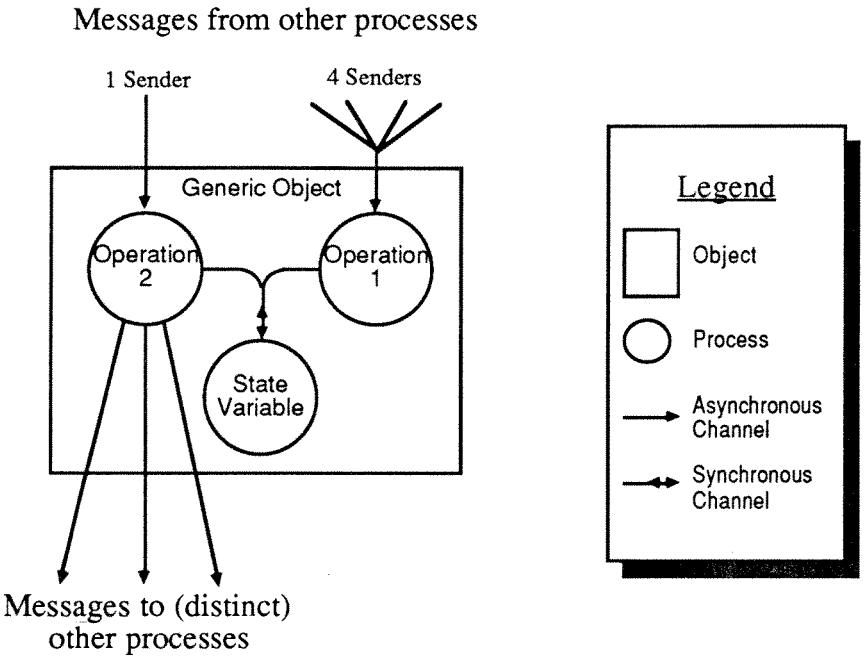
The language and semantics aspects of our work are most similar to that of the ESTEREL project. Like them, we intend to build our system upon a formal semantic framework that will allow temporal reasoning on program texts. We differ in that our semantics are not tightly coupled with a sequential model of computation such as a finite state machine. As a

result of this, we can write familiar classes of programs such as cyclic executives, and buffer managers that are either awkward or impossible to write in ESTEREL. Additionally, our semantics allows a straightforward language implementation on non-uniprocessor architectures.

3. Language Description

3.1. Programming Model

The language itself follows an object-oriented paradigm. Abstractly an object is a data repository with a set of operations for manipulating the data. Manipulations of the object's state are performed by the object itself based on the receipt of messages from other objects. A message is a typed communication along a named channel. The active agent inside an object that processes messages of a given type is called a process. An object may contain multiple processes, one for each message type (operation) that the object receives (implements). In addition, an object may contain processes that implement operations not visible to other objects. These "internal" operations are also invoked by sending a message on the appropriate channel. Objects may be nested. A simple object is shown graphically in Figure 1. Each component in this figure will be explained in more detail below.



This diagram illustrates a design for a generic object that implements two operations. The processes that implement these operations are labeled *Operation1* and *Operation2*. The object alters its state by sending messages to an encapsulated state object. Note that every channel has only one receiver.

Figure 1

In our model there is a one-to-one correspondence between channels, processes, and operations. For example, a channel uniquely identifies a process (the channel's receiver process) and an operation. In our world, everything from simple program variables to

complex programs, can be represented as objects. For the present, we will speak mainly of processes and messages rather than of objects and operations.†

A channel may have many writers (senders) but only one reader (receiver). There are two classes of channels: synchronous and asynchronous. Sending a message on a synchronous channel is equivalent to performing a procedure call from which typed values may be returned. Sending a message on an asynchronous channel allows the sender to proceed in parallel with the receiver. In the asynchronous case, the sender *never* waits - in any sense - for the receiver. A process may only receive messages of one type (from one channel). A process is activated by the receipt of a message. Once activated it may perform computations including sending messages to other processes. However, during an instance of a process's activation, it may send *at most one message* to any *specific* process. When it has finished its computations it deactivates and awaits its next message. An analogous view of a process is as a server. Inside of a process, the sending of messages may be controlled by data dependent logic. A schema for a process's behavior is given in Figure 2.

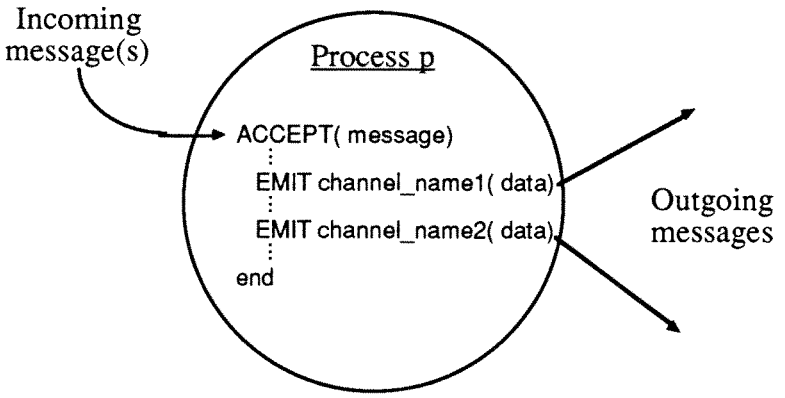


Figure 2

We say an object is "real-time" (or it behaves in real-time) if it is *always* ready to receive all messages that are sent to it. More specifically, all processes within a real-time object must always be ready to receive all messages that are sent to them *as they are sent* (i.e., without buffering). This property of objects is captured in the formal semantics of our language. For a program to be considered correct, all objects must behave in real-time. As it should be clear that it is always possible to write a program that could not adhere to its semantics for a particular implementation; it is the goal of the temporal verification procedure that follows to determine if in fact all processes in a program will behave in real-time for a given execution environment.

Our process model is based on a general characterization of the real-time environment as a set of processes external to the computer that forbid any type of computer initiated flow-

---

† While we have chosen to present our model from within the context of an object oriented domain, the object orientation in our model is not central to our main contributions: the analysis techniques that follow. What is important are our message passing paradigms. Hence for the remainder of the paper, we will concentrate mainly on the properties of processes and messages rather than of the related concepts of objects and operations.

control of their inputs or outputs. The implication of this is that internal computer processes that interface with the external world must always be ready to process an input from an external process. Stated another way, they must process their inputs in a time frame determined by the external world. Processes in the outside world are abstracted as objects that sporadically emit "messages" to objects internal to the system. It is a fundamental assumption that the worst case frequency with which external objects can send messages is known. We will require this information to be in the form of a minimum separation time between the emission of any two messages of a given type by a given external process.

Aside from the ACCEPT and EMIT constructs, the processing that takes place inside a process can be described by almost any high-level sequential programming language. We only require that *all* statements in this language take a deterministic amount of time to execute and that their worst case execution time should be statically determinable. As there are no constructs or schemas in our model which can cause logical blocking (in the sense of the language's semantics given below), we can determine a priori a worst case processing time for any message type

3.2. Logical Semantics

We further propose to develop a suitable semantic model that incorporates the passage of time. Our notion of time here is discrete time as measured by the arrival of messages. Each process has its own meta-clock. The arrival of a message at a process is a "tick" of the process's clock. Relative to this clock, the process's computations take "no time". For example, an implementation of each process on a dedicated, infinitely fast processor would always give the correct temporal semantics (assuming the computation terminates). Thus in spite of our terminology, from the user's standpoint, our language is logically a synchronous one. When messages are sent, they are instantly received at their destination and processed. Hence changing all asynchronous channels in a program to synchronous ones yields a semantically equivalent program. The difference between these programs lies in the fact that for a given implementation environment, we may be able to verify the temporal correctness of the former program but not the latter. The reasoning for this will become apparent in Section 4.2.

Our model is new in that not all process's concept of time need be related. Processes that do not interact directly or indirectly may have *independent* clocks. These clocks will be unsynchronized and need not even be "running" at the same rate. At the semantic level, there is no need for a global clock.

3.3. An Example

Consider the following simple stopwatch example. A hardware timer produces periodic interrupts with known frequency which are used to keep time in the expected manner. A button is also connected to the system to start and stop the stopwatch. An odd numbered push of the button starts the stopwatch while an even numbered push of the button stops the stopwatch and displays the elapsed time. A possible design for a program that performs these functions might have the structure as shown in Figure 3 below.

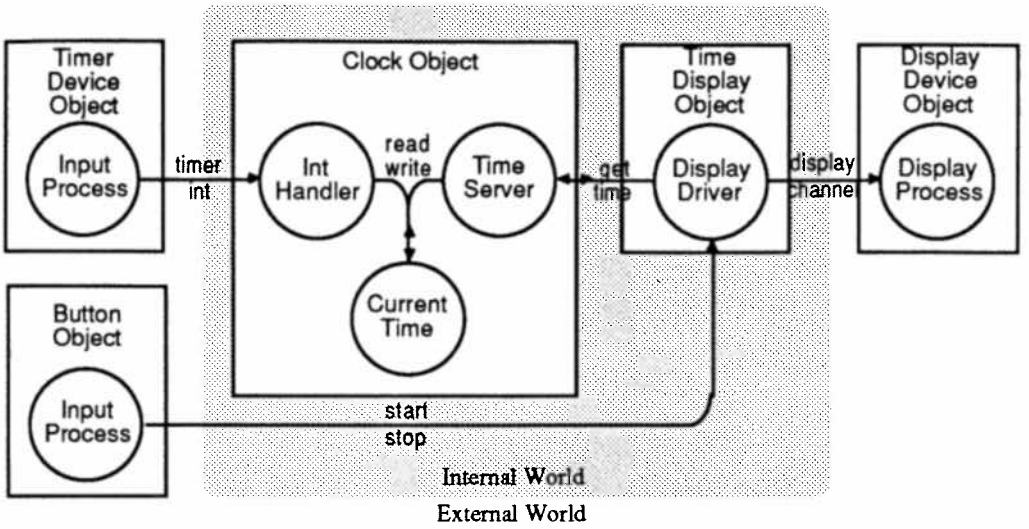


Figure 3

The external timer object produces messages of type *timer int* which are consumed by an internal clock object. The clock object is composed of three processes. Process *InterruptHandler* receives the timer messages and performs whatever timer setting or resetting is required. It then synchronously sends a message along channel *read/write* to the process *CurrentTime*. Process *CurrentTime* controls access to the storage object(s) that actually represents the value of the clock. In this case, process *CurrentTime* decodes this message, determines that it should update the time, increments its internal counters, and returns a status value. Upon receipt of a reply, process *InterruptHandler* terminates. The time-display object manages the display device and implements the stopwatch function. It receives messages from the external button object along channel *start/stop*. For each message received it gets the current time by synchronously sending a message to the process *TimeServer*. *TimeServer* gets the current time from process *CurrentTime* in a reply to its message sent on channel *read/write*. Next, depending on its current state, the time-display object either records the current time or computes the difference in time since its last activation and sends this value to an external display device on channel *display channel*.

This simple example motivates several important questions related to real-time. First, we may wish to know if the clock object will faithfully keep time. Our semantics tell us that all messages sent are instantly received and processed. Hence in the above example, it is not possible for the clock object to "miss" a *timer int* message sent from the Timer device. Thus the issue of whether or not the clock object correctly keeps time is reduced to an examination of whether or not the clock object implements a correct algorithm for computing the value of time given a *timer int* message. The same logic allows us to conclude that every button press will be recognized by the system.

The questions concerning the processing of messages from external objects deal with what has been called the *temporal correctness* of the system [Shaw et. al. 85]. For systems constructed using our paradigms, many of the temporal correctness issues have been abstracted out of the discussion by our semantic model. However, note that there are other issues of *logical* correctness that also involve "real-time" but cannot be answered by the language semantics directly. For example it is not clear at this point what the precision of

the stopwatch is. Although we will not discuss it, this can be determined by techniques employed during the temporal verification process that follows.

4. Temporal Verification and Run-Time System Design

The goal of the temporal verification process is to determine if under *all* circumstances, an implementation of a system constructed by following our model will be faithful to its semantics. By all circumstances we mean under all conditions wherein messages from the external world do not violate their stated worst case separation times. The result of this portion of the development process is a boolean answer. The system under examination will either always be semantically correct or it may not be. It is believed that this verification process can be shown to be optimal in the sense if there exists a method for correctly implementing an instance of our model, then the temporal verification process will succeed on that instance.

Temporal verification takes place during the semantic analysis of the program. There are essentially two classes of problems it is trying to solve. The first deals with "temporal programming errors" that are independent of a program's implementation (static analysis). The second addresses the run-time system's ability to schedule an implementation of the program on a set of processors so that the semantics of the program can be realized (dynamic analysis).

4.1. Process Graph Construction and Static Analysis

For both analysis problems, the program is represented by its process communication graph. The process communication graph is derived from the the structural diagram of the system (e.g., the figures constructed using our graphical design notation) by striping away the object boundaries and replicating channels with multiple senders across each sender. For example, the process graph for our simple stopwatch is shown below in Figure 4. Nodes in the graph represent processes and edges represents message paths (channels). Edges are labeled with values that represent the worst case (fastest) message transmission rate possible on the corresponding channel. Initially, the graph contains only message transmission rate information for channels whose senders are external processes. These initial values r_i, are computed from the minimum message separation times, p_i, that were required to be provided for each external object (device).

The first step in the static analysis is to label all edges in the graph. This is done in two phases. First, for each process the code is analyzed to determine its worst case transmission rate on all channels that it sends messages. This rate will be expressed as a function of the process's activation rate. Note that since processes are restricted to emitting at most one message per channel per activation, these functions $f(x)=y$ are bounded above by the identity function $I(x)=x$, and below by the zero function $Z(x)=0$. In the second phase, activation rates are determined by starting at the sources of the graph - external processes whose transmission rates are known - and evaluating the transmission rate function(s) for each node we encounter as we traverse the graph.

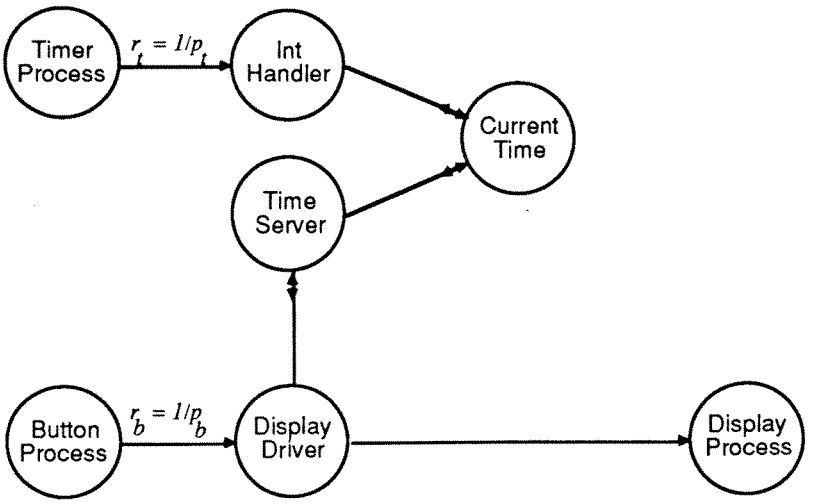


Figure 4

For example, with the exception of the display driver process, each process in the stopwatch example emits one message during each activation; the transmission rate function for each process is therefore the identity function. Recall that we (arbitrarily) designed the stopwatch to display the elapsed time every other button press (one press starts the watch, the next stops it and displays the time). Therefore the display driver process emits one message to the time server process on every activation but only one message every other activation to the display process. The complete process graph for the program is as shown in Figure 5†.

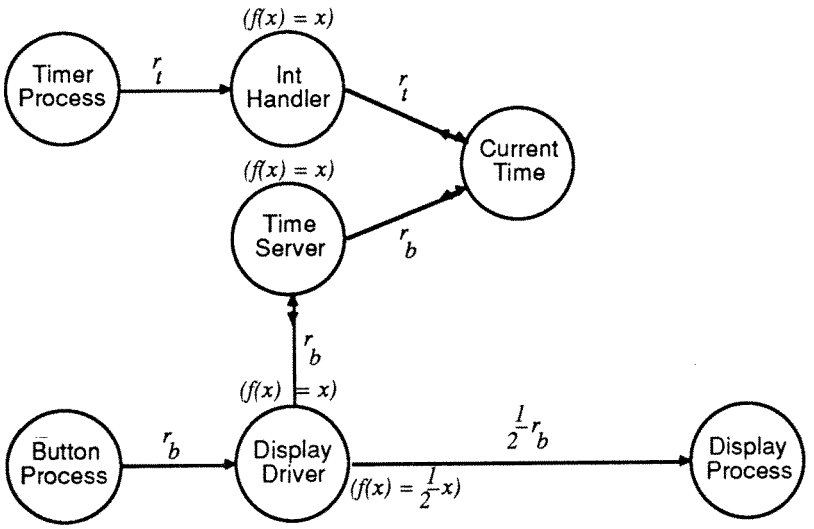


Figure 5

---

† Given this graph, one can determine bounds for the precision of the stopwatch. However, this analysis is beyond the scope of this paper.

Now consider the more complex example shown below in Figure 6. This is a design for a simplistic air traffic monitoring system. Radio messages announce the arrival of new aircraft into the airspace. The *Dispatcher* object receives these messages and activates one of two *Tracker* objects. Once activated, a tracker queries a *Radar* object to get positional information on its aircraft. This information is deposited in the *Track File* data base object. It is possible that a response from the *Radar* object may indicate that more than one aircraft has been found in a tracker's airspace. In this case the tracker informs the *Dispatcher* that a UFO has been found. The *Dispatcher* will then assign another tracker to the UFO. Periodically, a *Display Driver* object is awoken and will display information from the *Track File* on a display. This system has been designed to track no more than two aircraft simultaneously.

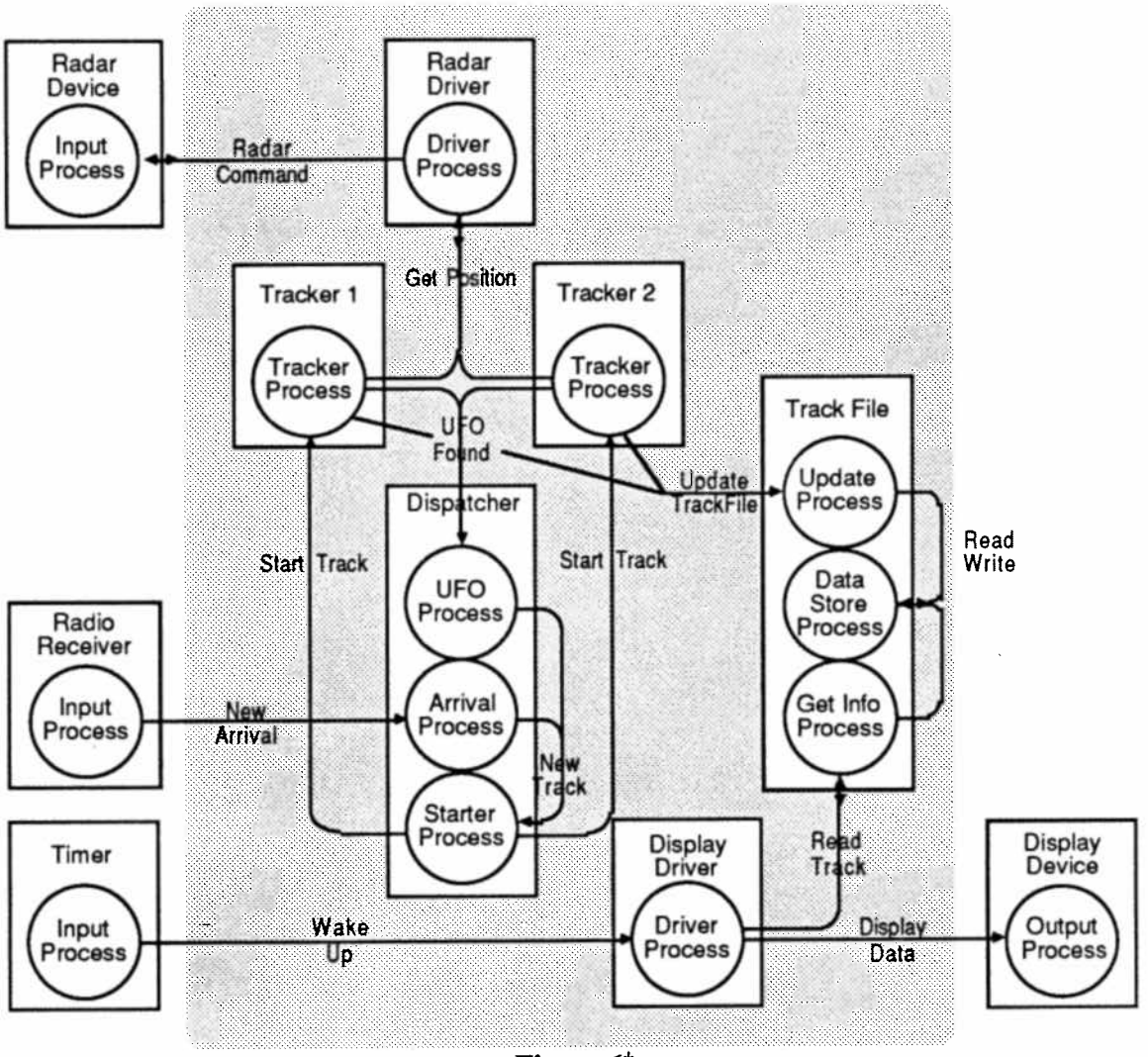


Figure 6†

† The fact that the internal channels *NewTrack* in the *Dispatcher* object, and *ReadWrite* in the *TrackFile* object, have been drawn as leaving and then re-entering the enclosing object is *not* significant.

The process graph for this example is shown in Figure 7. The temporal analysis is now more complex since there is a cycle in the graph that includes nodes *Starter process*, *Tracker process*, and *UFO process*. It is possible that given the code of these processes, there may be no unique solution to the transmission rate function along the channel from *UFO process* to *Starter process*. If this is the case then we say the cycle is unstable and the program will be rejected due to a *temporal programming error*. The programmer has implemented the temporal equivalent of an infinite loop: a *feedback loop*.

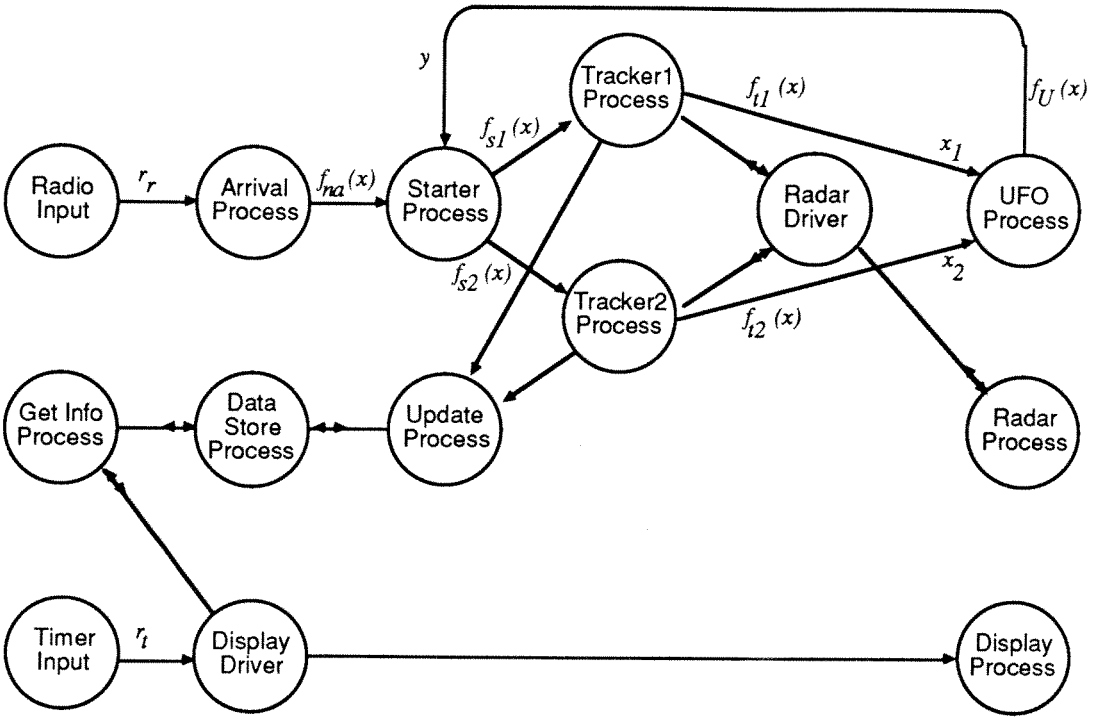


Figure 7

To determine if the cycle is stable, we compute the transmission rate functions as before. Note that for processes such as *Starter* and *UFO* that have multiple senders, their activation rate is the sum of their senders' message transmission rates. In this example we need to determine if the equation

$$y = f_U(x_1 + x_2)$$

$$= f_U(\, f_{t1}(\, f_{s1}(r_r+y)) + f_{t2}(\, f_{s2}(r_r+y))\,)$$

has a unique solution. In this example we may reasonably assume that $f_{s1}(x) = f_{s2}(x) = \frac{1}{2}x$, and that $f_U(x) = x$. This leaves us with y expressed in terms of the worst case expected rate with which UFOs are found. Further assuming that $f_{t1}(x) = f_{t2}(x)$ (call this $f_t(x)$) we have

$$y = f_U(\, 2f_t(\, \tfrac{1}{2}(r_r+y))\,)$$

$$= 2f_t(\, \tfrac{1}{2}(r_r+y)\,)$$

This equation in fact can be estimated by

$$y \leq 2I(\tfrac{1}{2}(r_r+y)) = 2(\tfrac{1}{2}(r_r+y))$$

$$\leq r_r+y$$

Thus if $f_t(x)$ is the identity function, we have $y = r_r+y$ which has no solution since we assume r_r is non-zero. Hence for the given transmission rate functions, it can be shown that the cycle will be stable if $f_t(x)$ has constant slope and is less than the identity function. For example if we have $f_t(x) = \frac{1}{100}x$, then $y = 2(f_t(\frac{1}{2}(r_r+y)))) = \frac{1}{100}(r_r+y)$ which is just $y = \frac{r_r}{99}$. We can see that in addition to identifying temporal programming errors, this rate analysis can also be used to determine theoretical bounds for rate functions that correspond to processes whose expected behavior may be unknown.

Note that had we treated $f_{t1}(x)$ and $f_{t2}(x)$ separately, the analysis would have been slightly more complex since we would have had to now solve the system of linear equations (assuming $f_U(x)$, $f_{s1}(x)$, and $f_{s2}(x)$ as before):

$$x_1 = f_{t1}(\tfrac{1}{2}(r_r + f_U(x_1+x_2)))$$

$$= f_{t1}(\tfrac{1}{2}(r_r+x_1+x_2)),$$

$$x_2 = f_{t2}(\tfrac{1}{2}(r_r + f_U(x_1+x_2)))$$

$$= f_{t2}(\tfrac{1}{2}(r_r+x_1+x_2)).$$

A final class of cycles that are disallowed are those involving only synchronous messages. This corresponds to indirect recursion and is disallowed to simplify the following execution cost analysis discussed below. If recursion is desired then it is suggested that the programmer implement it as an iteration. It is conceivable that this restriction could be relaxed in the future with the addition of syntax to avoid unbounded recursion.

4.2. Run-Time Analysis

This portion of the temporal verification process is a decision procedure that determines if it is possible to schedule the program represented by the process graph so that all processes will receive all messages that are sent to them. This decision procedure will necessarily require detailed information about the run-time system, the target architecture, and the timing properties of processes external to the system. Throughout this section we assume a shared memory architecture.

The run-time analysis has its foundations in deterministic scheduling theory. At this stage a program is represented as a network of processes that activated periodically with (assumed) known worst case cost. Our language's paradigm of processes and channels leads to a complete specification of the entire scheduling problem by four classes of problems. These are: the *single producer/single consumer* (SP/SC) process problem; the *multiple producer/single consumer* (MP/SC) process problem; the *synchronous single*

producer/single consumer (sync-SP/SC) process problem; and the *synchronous multiple producer/single consumer* (sync-MP/SC) problem. The first two problems deal with determining if a group of processes meet stated conditions for schedulability. The last two problem deals with an aspect of determining an upper bound for a process's execution time.

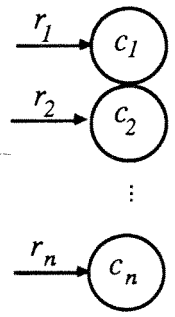
An SP/SC process is one which is activated by only one asynchronous sender. An MP/SC process is one which is activated by multiple asynchronous senders. For example, in Figure 6, the *Starter*, and *UFO* processes in the *Dispatcher* object are MP/SC processes, while the *Tracker* processes are SP/SC. Similarly, a sync-SP/SC process is one which is activated synchronously by only one sender and a sync-MP/SC process is one which is activated synchronously by multiple senders. In figure 6, the *Data Store* process in the *Track File* object and the *Driver* process in the *Radar Driver* object are sync-MP/SC processes while the *Get Info* process in the *Track File* object is a sync-SP/SC process.

4.2.1. Schedulability Analysis

The entire SP/SC process group can be more formally characterized as a set $T_{SP/SC}$, of n independent processes, each with known worst case execution time c_i, and activation rate r_i as shown in Figure 8. The determination of the worst case execution time for a process is addressed below. The tasks in $T_{SP/SC}$ can be mapped onto an equivalent set of periodic tasks with uniform periods. The simplest optimal solution to this problem is to employ n processors and dedicate a processor to each process. In this case the processes are schedulable if and only if $p_i \geq c_i$ for all i in $T_{SP/SC}$ (where $p_i = \frac{1}{r_i}$). Given fewer than n processors, we note that this task set can also be scheduled optimally (in the sense of [Mok 83]) on a set of shared processors using the unit-time preemption, least laxity algorithm. In this case the processes are schedulable if and only if

$$\sum_{i=0}^{n} \frac{c_i}{p_i} \leq k,$$

where k is the number of processors. We have been working on a refinement of this algorithm which should lower its overhead.

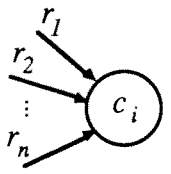


$$T_{SP/SC} = \{(\tfrac{1}{r_1}, c_1), (\tfrac{1}{r_2}, c_2), \ldots (\tfrac{1}{r_n}, c_n)\}$$

Figure 8

For the MP/SC process group we will deal with each process separately. Each process, t_i, in the MP/SC process group may be formally characterized as a set $T_{MP/SCi}$ of n identical, dependent processes with known worst case activation rate and identical worst case

execution time as shown in Figure 9. Note that these n processes are all conceptually executing the same code. Hence to ensure the consistency of the message consuming object, processing of a message at a MP/SC process will constitute a critical section. An implication of this is that this scheduling problem is an inherent uniprocessor problem. Adding additional processes to this problem will not improve its performance. For this reason we have limited our attention to uniprocessor algorithms. Presently we have developed an algorithm to schedule these processes and are actively working on a proof of its optimality.



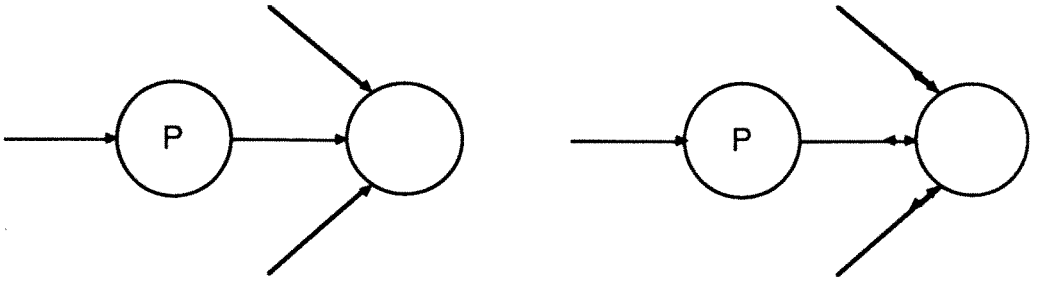
$$T_{MP/SC_i} = \{(\frac{1}{r_1}, c_i), (\frac{1}{r_2}, c_i), \ldots (\frac{1}{r_n}, c_i)\}$$

Figure 9

4.2.2. Cost Analysis

In both the single and multiple producer process groups above we have stated that their worst case execution time was known. Part of the execution time, c_{code}, is determined by the sequential code contained within the individual process. We intend to derive the worst case execution time for this code from bounds on program statements. Recall that we required all statements in the sequential language employed to have statically determinable execution times. The remainder of an SP/SC or MP/SC process's execution time, c_{delay}, is the time spent waiting for the reply to messages sent synchronously to other processes. If a process sends a message to a sync-SP/SC process, then one may compute the waiting time of the sender by "in-lining" (inserting) the sync-SP/SC process's code into that of the sender. The sync-MP/SC process problem is accounting for the delay incurred when sending synchronous messages to processes which have multiple senders. This problem is also currently under investigation.

When presenting the semantics of our message passing system (Section 3.2) we remarked that synchronous and asynchronous channels were semantically equivalent. The effect of using one channel type versus the other can be seen by examining the execution time cost for the channel's sender(s). Consider the scenario shown in Figure 10. In either case, irrespective of the channel type, process P has an execution time component c_{code}. In the asynchronous case, the total execution time cost for P is just c_{code}. However, for the synchronous case, the total execution time cost for P has the additional c_{delay} component. This additional cost in the synchronous case may cause the decision procedure from the previous section to fail. This is the basis for our remark that one may not be able to verify a previously verified program whose asynchronous channels have changes to synchronous one.



a) Asynchronous case b) Synchronous case

Two semantically equivalent programs.

Figure 10

5. Research Plan

The main focus of our research in the near future will be on refining and formalizing our programming model. To demonstrate the practicality of this work, we also plan to construct a small prototype programming system. To accomplish these goals we have laid out the steps listed below.

5.1. Programming Model

- Model Refinements - More study of the foundations of our programming model is needed. A more in depth analysis of related notations such as timed petri nets, data flow, and single assignment languages is in order.

- Model Expressiveness - Throughout the course of this research it will be incumbent upon us to demonstrate that our programming notation is convenient and expressive enough to construct programs of reasonable length and complexity. At a minimum we believe that one should be able to construct programs, subject to the constraints noted in Section 4, that can be expressed as general graphs. Indeed we have experimented with paper designs for programming language solutions to a variety of producer/consumer problems which we feel are endemic to many real-time applications.

- Optimality Assertions - Our model has motivated a particular decomposition into a set of scheduling problems. We need to show both that this decomposition is correct and that if we are unable to solve these scheduling problems for a particular instance of the model, then no other decomposition can satisfy the semantics of the model. An immediate first sub-problem to address is to complete the optimality proofs for the scheduling algorithms developed for Section 4.2.

5.2. Prototype Construction

Although the proposed language system could be implemented on almost any conventional architecture, we have chosen a shared memory multiprocessor for a prototype implementation. There are several reasons for this. It is expected that our worst case style of temporal analysis will lead to systems with necessarily pessimistic performance

requirements. A potential problem with this is that applications may find that they cannot be verified for a uniprocessor implementation. One way to mitigate this problem is to exploit the parallelism in the language and employ several processors in the execution of the program. We have chosen a tightly coupled architecture because it is more amenable to existing optimal scheduling theory. This theory requires that processors always execute the globally optimal next job at each scheduling opportunity.

Although listed individually, each component of the research clearly interacts with other components. Thus rather than solving one problem completely, we will work incrementally on each in concert with the others.

- Language Design - While the fundamentals of the language have been established, other issues such as the syntax have not received much attention. For a prototype we will most likely adopt a rather restrictive syntax in order to simplify the verification process.

- Run-Time System construction - A run-time system that implements the scheduling algorithms from Section 4.2 needs to be constructed. Both the key feature and the key drawback of our run-time system will be its reliance on a central process ready queue. A central, ordered queue will be required in order to achieve a notion of optimality in our implementation. A worrisome problem, however, is that since we are scheduling processes that are as lightweight as, for example, a small procedure call, access to this queue may likely become the bottleneck of the system. Previous design work with a centralized multiprocessor scheduler has shown us that under certain circumstances efficient dispatching is indeed possible [Baker & Jeffay 86].

  A prototype run-time system is envisioned for the short term wherein each process runs on a dedicated processor.

- Compiler construction - It is envisioned that a compiler for our language could be rapidly constructed. Most likely this compiler would translate our language into a conventional high-level sequential language that would make procedure calls to our run-time system. Actual machine code would be generated by an existing compiler for our target language.

- Temporal Verification Processor - Recall that temporal verification occurs in two phases. In the structural verification phase, the verification processor must construct the process graph, determine the transmission rate functions for each process and look for unstable or recursive cycles. The second, machine dependent phase, must determine the (worst case) cost of processing a message for each process. Given the processing costs and message arrival rates, it must then apply a decision procedure specific to the scheduling algorithm used to determine if all tasks can be scheduled in real-time. Another dimension to the cost analysis problem is our ability to account for whatever overhead the run-time system imposes on programs.

- Simulation Experiments - Since we have used the same abstractions to speak about external processes as we have internal processes, a prototype multiprocessor system could use a combination of dedicated internal processes and hardware timers to simulate the behavior of arbitrary external devices.

6. Objectives and Contributions

We see our research as having the potential to make contributions in the following areas.

- A programming model capable of automatic verification of adherence of timing constraints. - Given a target machine and run-time system, programs can be temporally verified solely by an examination of their text.

- Design paradigms for representation of real-time problems - Our paradigms of real-time producer/consumer interactions could be used as a simple yet powerful notation for representing and reasoning about real-time problems and constraints.

- Creation of an optimal real-time computing environment. - If our verification procedure fails, then it is not possible to schedule a program with the same process structure under any discipline.

- Practical results in real-time scheduling theory. - The development of optimal and near optimal scheduling algorithms for general real-time process definitions in a multiprocessor environment.

- Empirical results of the practicality of employing the worst case analysis common in real-time systems. - This has many facets in our project. Among them are the ability to determine the worst case behavior of sequential code fragments (static analysis). We need also determine the worst case behavior of ordinary code in a shared memory multiprocessor environment (dynamic analysis). This would include accounting for such factors as pipelining, memory/bus contention and caching. A useful result would be to measure experimentally the deviation between worst case and average (experienced) case. A further investigation might lead to the formulation of architectural features or requirements for real-time computing based on worst case analysis.

7. References

[Baker & Jeffay 86]
Baker, T.P., Jeffay, K., *A Lace for Ada's Corset*, University of Washington Department of Computer Science Technical Report TR 86-09-06, October 1986.

[Berry & Cosserat 85]
Berry, G., Cosserat, L., *The ESTEREL Synchronous Programming Language and its Mathematical Semantics*, Lecture Notes in Computer Science, Vol 197 (1985) pp. 389-448.

[Kligerman & Stoyenko 86].
Kligerman, E., Stoyenko, A.D., *Real-Time Euclid: A Language for Reliable Real-Time Systems*, IEEE TOSE Vol. SE-12, No. 9 (Sept. 1986), 941-949.

[Lampson & Redell 80]
Lampson, B.W., Redell, D.W., *Experience with Processes and Monitors in Mesa*, CACM, Vol. 23, No. 2 (February 1980), pp. 105-117.

[Liu & Layland 73]
Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, JACM, Vol. 20, No.1 (January 1973), pp. 46-61.

[Mok 83] Mok, A.K., *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, MIT Laboratory for Computer Science Technical Report (Ph.D. Thesis) #MIT/LCS/TR-297, 1983.

[Shaw et. al. 85]
Shaw, A.C., Binding, C., Hu, W.L., Jeffay, K., *Research in Real-Time Systems*, IEEE Third Workshop in Real-Time Operating Systems, Boston, MA, February 1986 (also available as University of Washington Department of Computer Science Technical Report TR-85-12-05 (December 1985)).

[Shaw 87] Shaw, A.C., *Reasoning About Time in Higher-Level Language Software*, University of Washington Department of Computer Science Technical Report TR-87-08-05 (August 1987).

[US DoD 83] **Reference Manual for the Ada Programming Language**, ANSI /Military standard MIL-STD-1815A, US DoD, January 1983.

[Wirth 77] Wirth, N., *Toward a discipline of real-time programming.*, CACM Vol. 20, No. 8 (Aug. 1977), 577-583.