Issues in Applicative Real-time Programming*

by

Lennart Edblom and Daniel P. Friedman

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 129

ISSUES IN APPLICATIVE REAL-TIME PROGRAMMING

Lennart Edblom and Daniel P. Friedman

August 1982

*Research reported herein was supported in part by the National Science Foundation under grant number MCS 77-22325 and grant number MCS 79-04183.

D.1.1 [**Programming Techniques**]:

Applicative (Functional) Programming;

D.2.1 [**Software Engineering**]:

Requirements/Specifications - *languages;*

D.3.2 [**Programming Languages**]:

Language Classifications - *applicative languages;*

D.3.3 [**Programming Languages**]:

Language Constructs

General Terms; Applicative programming, real-time programming languages.

Additional Key Words and Phrases: Time-sensitive behavior, streams, hiatons, execution time predictability.

Table of contents

Issues in Applicative Real-time Programming

0. Introduction

In this paper we address some of the questions concerning real-time programming, especially from the standpoint of applicative programming. First we would like to identify the characteristic properties of real-time programs. What are the building-blocks they are composed of? Which requirements are they supposed to meet? We want to isolate the fundamental properties of real-time programming. When we have done that, can we make these properties transparent and interface them into an applicative framework? How can we do that? If it is impossible, why is it so? Which crucial obstacles prevent us from doing so? We do not claim to solve all these problems, rather we try to identify and state the fundamental problems concerning real-time programming. Then we try to apply tools invented by McCarthy, Landin, Friedman & Wise, Ashcroft & Wadge and others to solve real-time problems using an applicative style of programming.

In the first section we try to answer the question about the nature of real-time programming, and to develop an idea of which requirements a real-time system must meet. In the following section we investigate how this influences the design of high-level programming languages for real-time applications, which primitives and which operations that are needed in such a language. In particular, we take a look at a proposed standard for real-time FORTRAN [Heller et al. 78]. We also consider why it is at all desirable to develop good high-level languages for real-time programming.

Most real-time systems are naturally expressed as a number of components, operating in parallel with some degree of interaction. Consequently, we need tools to express parallel processing, synchronization, and communication among tasks. It is therefore appropriate to review an approach to applicative multiprogramming. That is done in Section 3, where it is shown how communication and synchronization can be imple-

mented by means of *streams* [Landin 65], and how the problem of handling asynchronous events can be solved using the multiset constructor *frons* [Friedman & Wise 80].

The centerpiece of the paper is Section 4, where we tackle some of the problems concerning applicative real-time programming. The fundamental question is: Is it possible to express time-sensitive computations in an applicative style? We claim that it is possible, and present a tentative approach to how it may be done by including objects representing time, called *hiatons* [Faustini 82] , in the communication streams. We also discuss the possibility of estimating the time needed to execute a sequence of code in an applicative language, something which often is necessary to do in real-time programming.

Finally, Section 5 argues that applicative languages may indeed be a useful tool in real-time programming. They enable a programmer to write programs on a high level, programs which will be easier to prove correct than today's programs. When verified, they can be transformed into programs more suitable for implementation and execution.

The reader is assumed to be familiar with the functional style of programming. If not, consult [Henderson 80] or [Backus 78] for an introduction.The applicative programs in this paper are to a large extent written using a syntax similar to that of [Henderson 80].

1. What is real-time programming?

An ordinary sequential program is designed such that the computed results are independent of the execution speed of their processors, or of any other external events. To speed up computation one may use several concurrently operating processors. Pro-

grams using several processors consist of several routines, called processes, which are themselves purely sequential and are executed concurrently. In existing programming languages, communication between processes is accomplished via some kind of shared variables and synchronization signals. However, such multiprograms still specify the computed results independently of the absolute and relative speeds of employed processors. But suppose that certain processes - which are not programmable - may fail to wait for synchronization signals. For example, the nonprogrammable process may be a human, unwilling to wait for the results of his request longer than a certain duration. We have then entered the field of *real-time programming* where the results of our programs, and even more important, their validity, depend on the execution speed of the utilized processors, and/or on external events.

The crucial property that distinguishes real-time programs from other programs is accordingly that they are *time-dependent* , or *time-sensitive* , in one way or another. It is almost always required that a real-time system should respond appropriately to some kind of events within a certain time limit. In some systems, for instance an airline reservation system, the limit may be a few seconds. In other systems the constraints are much tighter. We might for example have a system that controls a chemical process. It is not unusual that the system wants to sample some output signals every millisecond. It must then be able to perform the necessary calculations and issue the correct feedback signal within a millisecond, otherwise in the end the whole plant may blow up. To make sure that our programs meet such time constraints, we must be able to calculate how long time that is needed to run a certain piece of code, to calculate the cost of an expression. One must know beforehand if the piece of code he's just written will execute in less than a millisecond.

Not all time dependencies consist of tight constraints. It might be the case that a system should perform a certain operation every minute. Now the problem is hardly to find the time to do this, but rather to know when a minute has elapsed. One way is to have some kind of clock or timer, which generates an interrupt each minute. Obviously, one advantage with the interrupt mechanism is that it enables us to write and execute other tasks in the system without any consideration of the task that should be performed each minute. But two clocks are never exactly synchronous, and even if it wouldn't matter in this particular example if we had to change to a slightly different clock, there might exist systems where the exchange of a clock for another might introduce errors in the timing of some components, and possibly invalidate the whole system. But if we want to manage without a clock, we again have to know how long it takes to execute each particular piece of code that may be executed during this minute, in order to be able to know when a minute has elapsed.

One of the reasons people use assembly languages and real-time FORTRAN is that in those languages you know the time required to execute each particular statement. In Section 4 we discuss the possibility of writing applicative programs whose execution times are predictable.

Another aspect of time in real-time systems is that a real-time system must be able to respond appropriately to external events that appear asynchronously, at indeterminate times. The handling of these asynchronous events must not interfere with other computations in progress, but must be properly synchronized and interleaved with them. An approach to this problem will be discussed in Section 3 about applicative multiprogramming.

Suppose however that the handling of these asynchronous events is subject to some time constraints, i.e. if an event has occurred within a certain period of time, do something, otherwise do another thing. Then we have got a situation where the rate of the input effects not only the *rate* of the output, but it effects *what* is output! This is not a functional behavior. But recall that the most characteristic property of applicative programming is its functionality, that is, the output of a function is determined solely by its input, and there are no side-effects. However, this situation violates that property. This raises the question: Can all types of real-time behavior and all real-time computations be expressed functionally, in an applicative language? If we want to use applicative languages for real-time programming, we must find an affirmative answer to this question. In Section 4 we adress this problem, and propose a solution by introducing a special kind of object, called *hiaton* (from "hiatus", meaning a pause; according to [Faustini 82] the term is due to W.Wadge and E.A.Ashcroft). A hiaton represents a unit of time, and timing information is encoded by allowing hiatons as elements of the streams used for process communication.

We claim that what we have discussed so far are the most important properties of real-time programming, the most fundamental building-blocks of real-time systems. There are, of course, also many other requirements on real-time systems. They are however of a more general nature, concerning reliability, efficiency etc., and are therefore outside the scope of this paper. A good overview of these questions can be found in [Dreisbach & Weissman 77]. [Infotech 71] and [IFAC 78] are other references with papers that discuss the characteristics of and requirements on real-time systems.

2. Real-time programming languages

In this section we present the requirements for a real-time programming language. What features are essential to express real-time behavior, and what capabilities are desirable? In the second part of this section, we take a look at the proposed standard for real-time FORTRAN, and try to draw some conclusions concerning the language requirements for industrial real-time applications.

Naturally, there are many, sometimes widely differing, opinions about the ideal structure of a real-time programming language. One good discussion of what properties such a language should have can be found in [Dreisbach & Weissman 77]. Additional opinions are presented in [SPL 74a],especially [Barnes 74], [Wirth 77b] and [Ichbiah et al 79].

2.1. Language requirements for real-time programming

In the past, real-time systems have to a large extent been programmed in assembly code. High level languages have only been used in the design phase, and for the less time-critical parts of the system. There are several reasons for this:

Predictability. As discussed in the previous section, it is sometimes necessary to be able to calculate in advance bounds for the time needed to execute a certain code sequence, to make sure that the program meets all time constraints. This is much easier to do for assembly code programs.

Efficiency. In time-critical parts, compilation of high-level code would simply not yield machine code that was efficient enough.

Machine dependencies and communication. Existing high level languages have provided no facilities to deal with machine-dependent features, for instance to access the hardware registers. Neither has it been possible to express the details of process communication and synchronization in an adequate and precise way, so the programmer has been obliged to resort to assembly languages.

However, the increasing size and complexity of real-time systems has made it clear that it is not feasible to use assembly languages. Some kind of high-level language is needed. Naturally, this need has induced efforts to design special real-time high-level languages. One example is RTL [Schoeffler 71], and its successor RTL/2, [SPL74a],[SPL74b], which is widely used in the UK and Western Europe. Another example is Modula [Wirth 77a], [Holden & Wand 77]. Since we in this paper occasionally will use Modula in examples of conventional real-time programming, the reader with no knowledge of Modula is encouraged to read [Wirth 77a].

We will now try to point out the most important requirements on a high-level language for real-time programming. We start by noting that although real-time programming conceptually is different from multiprogramming, it is nevertheless the case that the concept of parallelism is intrinsic to the nature of most real-time systems. Such systems can often be thought of as several tasks, operating in parallel with some degree of interaction. Therefore, it is necessary that real-time languages include features that support parallelism.

According to [Wirth 77b] the most important single item in a real-time programming language is a notational unit for describing processes that are themselves purely sequential, but can be executed concurrently. The reason for this is that 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. The third item is an object to trigger continuation after waiting. Put another way, we need features for communication and synchronization among processes.

These requirements can be fulfilled in an applicative language. A function definition in a functional program is certainly an adequate notational unit for describing a sequential process that can be executed concurrently with other processes. Communication and synchronization is achieved, not with the help of shared variables and signals, but using streams. A *stream* ([Landin 65]) is a (possibly infinite) list, where the first elements can be accessed before the rest is defined. A stream can be thought of as a communication channel between two processes, one which produces the elements of the stream, and another which consumes these elements. In Section 3 we will explain in more detail how streams can be used instead of shared variables and signals.

A good real-time language must also have capabilities to access and make full use of hardware, peripheral devices, and other macine-dependent capabilities. However, the machine-dependent parts of a program should be explicitly encapsulated and separated from the rest of the program. In Modula, this is accomplished by means of *device modules*. There doesn't seem to be any conceptual obstacle to making a similar construction in an applicative language, although no closer investigations have been made.

As mentioned in the previous section, real-time systems are usually large and complex. This makes it very important that we use a programming language that helps to master this complexity. In other words, it must be simple, clear, and well structured, in order that programs written in it will be easy to write, debug, validate and maintain. In agreement with this, [Pike71] states that among the goals of realtime language design efforts should be:

1. Lucidity
2. Freedom from side effects.
3. Enforcement of programming discipline.
4. Natural modularity.
5. Ease of learning and use.

These requirements could equally well serve as a description of an applicative programming language. Most programmers that are acquainted with the functional style of programming agree that algorithms can be more clearly and concisely expressed in an applicative language than in any of the existing imperative languages, because an applicative style adheres more closely to the way we think about problems, and the way we formulate them in natural language. For the same reason, applicative languages are also easier to learn and to use. Freedom from side effects is an inherent property of applicative languages, and because of its dependence upon functional composition, applicative programming necessitates thinking in terms of top-down stepwise refinement, and provides natural modularity. We conclude that if we could enrich an applicaive language with the necessary primitives and operations for a real-time environment, it would be well suited for real-time programming.

2.2. Language requirements for industrial real-time applications

In [Heller et al 1978] the requirements on a programming language for industrial real-time applications are defined in terms of FORTRAN procedures which realize the necessary operations. These procedures can be divided into five groups, each one handling a separate type of function. These groups are day and time information, multiprogramming, binary pattern and bit processing, process I/O and file handling.

The procedures of primary interest for this paper are those concerned with multiprogramming. They can be further divided into procedures for task scheduling, i.e for starting and terminating of tasks, and for task synchronization. There are different proposals concerning the details of the procedures, but a real-time programmer will need at least the following operations for starting of tasks:

- start a task immediately or after a specified time delay
- start a task at an absolute time
- execute a task when a specified event occurs, i.e we must be able to respond properly to events which happen at indeterminate times.
- start a task immediately or after a specified time delay, and execute it periodically with a specified time interval between executions.
- start a task at an absolute time and execute it periodically.

Next we need some operations for termination of tasks; specifically we must be able to

- terminate tasks waiting for time (i.e stop future execution of tasks scheduled to start at a certain time)
- terminate tasks waiting for events
- remove a task from the system

The procedures for task synchronization manages synchronization with respect to events (interrupts) and times, and intertask synchronization. For event/time synchronization we need the possibilities to

- wait during a specified time delay
- wait for a specified event (but not more than a maximum time, to avoid deadlock)

Finally, for intertask synchronization we must be able to

- suspend a task until a certain condition is true, e.g until another task has produced a needed value
- cause the resumption of a task when a certain condition is true.

From the above we again conclude that in a real-time programming system it is essential to have a concept of time, to be able to schedule and synchronize tasks with respect to an external clock, or with respect to each other. One also needs the ability to specify the execution of a process at a certain time, or after a specified time delay, or to periodically execute a process. We may also want to synchronize two processes in such a way that whichever of them first reaches the "synchronization point" in its code, waits there until the other process reaches its corresponding "synchronization point". Finally, we must be able to handle events that appear asynchronously. In Section 4, we will see

how many of these requirements can be satisfied, using hiatons.

Since our primary area of interest in this paper is the time-sensitive part of real-time programming, we will not cover the other four groups of operations in this paper. The interested reader is referred to [Heller et al 78].

3. An approach to applicative multiprogramming

3.1. Output-driven computation

What language constructs are necessary to accommodate applicative multiprogramming? Let's first of all point out that in the particular applicative system, that we assume in this section, all computation is output-driven (other terms for this approach is demand-driven [Keller et al. 79], and lazy evaluation [Henderson & Morris 76]). As will be seen below, this property is essential in implementing process communication.

The output-driven approach means that a list can exist while none of its components are determined. The constructor *cons* is non-strict, it suspends evaluation of both its arguments.

A *stream* [Landin 65] is a list constructed by a constructor which is strict only in its first argument. The important property of streams is that the first element of a stream can be accessed, while the rest of the stream is undetermined. As was indicated in the introduction, this property makes it possible to use streams to implement communication between concurrent processes.

As an example, we will imagine two processes, P1 and P2 that produce some kind of values, let's for simplicity assume that P1 produces a list of the odd positive integers, and P2 a list of the even positive integers. A third process takes one value from P1 and

one from P2, performs some operations on these values, and produces a resulting value. To keep things as simple as possible, we will assume that this process just adds the two values together. Consequently we name the process Sum.

A program in Modula that runs these three processes concurrently will look like this:

```
module Sumlists
    interface module P1_to_Sum
        define Fetch 1, Deposit1; (*Names that may be
            accessed from outside the interface module*)
        var buffer:integer;
            empty: boolean;
            free, ready: signal;

        procedure Fetch1 (var x:integer);
        begin if empty then wait(ready) end;
            x:=buffer; empty:=true;
            send(free)
        end Fetch1;

        procedure Deposit1 (x:integer);
        begin if not empty then wait(free);
            buffer:=x; empty:=false;
            send(ready)
        end P1_to_Sum;

    begin (* initialize buffer *)
        empty := true
    end P1_to_Sum

      (* A similar interface module, P2_to_Sum, with procedures Fetch2 and Depo-
    sit2
        has to be declared and used for the transmission of values from P2 to Sum
    *)

    process P1;
        var n:integer;
    begin n:=-1;
        loop n:=n+ 2;
            Deposit1(n);
        end
    end P1;

    process P2; (*Similar to P1*)
```

```
    :
    :
  end P2;

  process Sum;
    var n1, n2:integer;
  begin
    loop Fetch1(n1); Fetch2(n2);
      write(n1+ n2)
    end
  end Sum;

begin Sum; P1; P2;
end Sumlists;
```

The three processes Sum, P1 and P2 are started in parallel. (If we only have one processor, we will of course have quasi-parallel execution). When Sum tries to Fetch1(n1) it has to wait until P1 puts an integer in the buffer (i.e. assigns a value to the shared variable), and issues the signal send(free). Likewise Sum has to wait until P2 has produced a value and deposited that value, before it can Fetch2(n2). On the other hand, P1 and P2 cannot deposit a new value in their respective buffers until Sum has fetched the previous value. The computation proceeds in this fashion, with process synchronization and communication achieved using signals and interface modules.

The corresponding applicative program will look like this:

```
let Sum A B = cons (add (first A)(first B)) (Sum (rest A)(rest B))
  and P1 X = cons X (P1 (add X 2))
  and P2 X = cons X (P2 (add X 2))
in Sum (P1 1)(P2 2)
```

(In the definition of Sum we assume that the streams A and B are infinite,so we do not have to test if either is empty).

How does the computation proceed when this program is executed? First of all we may note that since Sum uses *cons,* the resulting stream itself is not manifest. Only if we request that (part of it) be printed, will suspended references be coerced and values

be computed. So let's suppose that we want to print, say the first ten values of the result. Since this requires the first values of the results of P1 and P2, the demand-driven evaluation will cause evaluation of *first P1 1* and *first P2 2*. These are the only values that P1 and P2 will compute at this time, the rest of the streams are still suspended. Speaking in terms of control flow, the control will now return to Sum, which adds together these two values and delivers the result to the output driver. When the output device demands a new value, Sum needs the next value from P1 and P2, so they are computed and sent to Sum. The computation goes on like this, with Sum, P1 and P2 acting like coroutines. As in the Modula program, P1 and P2 produce only one value at a time. Only when Sum has consumed those values, and demands two new values, will P1 and P2 resume computation.

When we compare the two programs,we immediately note that the applicative program is much shorter. Of course this comparison is not entirely fair. Since Modula is a typed language, and our applicative language is not, we don't have to cope with the whole machinery of declarations, with the immediate consequence of shorter programs. But apart from that, there is still a significant difference between the two programs. It's much easier to express our intuitive notion of what is happening in an applicative language than in Modula. In Modula, we have to give a detailed specification of the communication protocols between two procedures, and we have to use signals in order to synchronize the processes. In contrast, all these functions are automatically performed by the applicative program by virtue of the output-driven computations in the system. In Modula, we also have to define shared variables and interface modules, and explicitly manipulate these variables. Using applicative programming on the other hand, there is no need to manipulate the streams that correspond to the shared variables. Since all computations are defined as functions, a stream can be the result of a function, and the

input to another function. All result values are automatically transmitted via the stream.

One might argue that the constructs in Modula are more powerful, since one may extend the buffer to contain more than one element by some small changes in the interface module. But the communication by streams isn't limited to one element at a time either. If one process demands n values from another process , then the next n values of the stream connecting these processes are computed! The only limitation is that values can't be computed in advance and stored in the buffer, as they can in Modula. However, if we have a multiprocessor system, one might imagine strategies to anticipate a demand and compute values in advance.

3.2. Handling indeterminism

Let's now modify our previous example. Assume that the two processes P1 and P2 produce their respective integers asynchronously, at indeterminate times. Since the values no longer arrive in pairs, we can't add them together, so we exchange the process Sum for another process, Merge, that merges these two streams into one in a timely fashion. That is, Merge shouldn't take one element from P1 and one element from P2, alternately, but should take whatever element that is ready.

This is an instance of one of the major problems in real-time programming, the problem of handling asynchronous and indeterminately occurring external events using a programming style built upon deterministic and synchronous interprocess communication. Different tools and constructs have been invented to grapple with this problem of timing, for instance semaphores [Dijkstra 68], monitors [Hoare 74] and signals and interface modules [Wirth 77a].

A new perspective on this problem is offered by [Friedman & Wise 80]. They introduce a new constructor, *frons* , which constructs an unordered structure, a *multiset* , in a manner similar to the way a list is constructed by the LISP sequence constructor *cons*. The elements of the structure constructed by *frons* are all *suspended*. A value is computed only if the suspension is probed by one of the access functions *first* or *rest*. When we ask for the first element of M, evaluation of all the elements of M start and proceed concurrently. When any one of the evaluations *converge,* the corresponding value is returned as the value of *first M*. (An evaluation is said to converge if the associated computation terminates in finite time, yielding a well-formed result).

By embedding the asynchronus events of a real-time system within a multiset, the programmer need not worry about flow of control on the arrival of external signals, since there is no explicit flow of control. All that is required of the programmer is to specify the contents of his multiset, in other words, what different events that may occur, and what to do when an element of the multiset is accessed, that is, when something has happened. He should make sure that his program gives valid results no matter which of the possible sequences appear.

We will now solve our modified problem, using our new tools: *frons* and multisets. The code for the Merge function will look like this:

```
Merge M =
    let (h . t) = M
    in if null M then NIL
        else if null h then Merge t
        else cons (first h)(Merge (frons (rest h) t))
```

where the notation *(x . y) = z* matches *x* to the first element (the *head*) of *z,* and *y* to the rest (the *tail*) of *z*.

We must also introduce the function *streamify*. *Streamify P* converts the list *P*, constructed by suspended *cons* to a Landin-stream whose first element always is manifest. It must be used to ensure that *first P* always returns a manifest element, and not a suspension.

Now the expression

Merge (frons streamify(P1) frons streamify(P2) nil)

solves our problem. When we ask for the first element of the result of this expression, evaluation of both P1 and P2 starts. As soon as one of P1 and P2 converges to a value, that value is taken as the first value in the resulting stream. The next value to be merged will be the next value that any one of P1 or P2 produces. The evaluation continues in this way, the elements will be merged in the order they converge, regardless of which process that produces them.

We might notice that the Merge function doesn't put any limit on the number of elements in M, Merge may accomodate any number of processes.

Let's return to the more general problem of handling asynchronous events. To solve this problem, we construct a multiset, containing all processes that cause these events. We must also create a function, corresponding to Merge, that takes care of the events that occur. Phrased in conventional terms, it would be called an interrupt handling routine. That routine would ask for the "first" object of the multiset. As soon as any of the processes has produced a value (that is, some kind of event has occurred) that value will be returned to the "interrupt handling routine" where it is handled in the appropriate way. Then we ask for first of the rest of the multiset, and so on. If two events happen simultaneously, they will nevertheless be merged in some order and handled one after another. Put another way, only one of the events may be chosen as *first*, and when that

one has been handled the other one will be chosen as *first* of the rest of the multiset and taken care of.

If we assume that we already have a database where it is recorded which process (processes) should start when a certain event occurs, we may now solve the problem of how to start a process when a certain event occurs. We embed the asynchronous events within a multiset as indicated above. When an event occurs the "interrupt handling routine" will look up in the database which processes are to be started and send a start message on the input stream of those processes. The solution will be similar to the solution of the airline reservation system in [Dennis 77].

The problem of handling asynchronously occurring events is also discussed in [Henderson 82]. Henderson defines a function "interleave", which performs the same task as our Merge function. Since he doesn't use *frons,* "interleave" is a fuction not only of the elements of the sequences which constitute its arguments, but also of the time at which each element becomes completely defined. Henderson goes on to discuss if "interleave" is a pure function, if it should be included in an applicative language, and how one could implement it in such a case.

4. Applicative real-time programming

If we are to be able to use applicative languages for real-time programming we must first of all solve the problem with the nonfunctional behavior of some real-time computations. In the first part of this section we will show how we may do this with a special kind of object called a *hiaton* [Faustini 82]. A hiaton can be thought of as a unit of delay that can be mixed with the regular data objects (also called *datons)* of a stream. Hiatons allow a function to produce something regularly even if it has no real output.

Using hiatons in our streams we can embed the timing information and time dependencies in the data structures, and preserve a purely functional behavior. We also discuss some of the new problems that are raised by the introduction of hiatons.

In Section 1 we also stated that it is a desirable property of real-time programs that we should be able to predict the time needed to execute a certain code sequence. In the second part of this section we discuss if and how this can be realized in an applicative language.

4.1. Programming time-sensitive functions applicatively

Let us consider another, more difficult version of the problem to add two streams together. Assume that the even integers arrive regularly, but the odd integers do not, there is a delay, sometimes fairly large, between each odd integer. Assume furthermore that the process that sums (or, generally, consumes) these streams is operating under a certain time constraint. Specifically, if an odd integer hasn't arrived within a certain time limit the corresponding even integer is thrown away (or output to a stream of "wasted evens")

How can we express a solution to this problem in an applicative language? Evidently, the result is time dependent. The resulting output will depend on which odd integers did arrive before timeout, and which didn't, because this in turn will determine which evens will be wasted, and which elements will be in the result stream . That means that the expression

Sum Evens Odds

may evaluate to different results if it is evaluated twice, even if it is evaluated in exactly the same context both times. The expression is no longer referentially transparent,

because of the influence of time.

This problem is certainly not artificial or very contrived. It also appears in industrial real-time programming. Suppose we have a task that controls some chemical process. Each second a device samples some process variable and sends the result to the controlling task. This corresponds to the stream of even integers. As long as the chemical process runs without problems nothing happens, the sampled values are simply thrown away by the controller. But if something goes wrong, then an interrupt occurs. This corresponds to something arriving on the stream of odd integers. The controlling task now must access the latest value of the control variable (and presumably some more control variables and other information, too) and decide on a suitable action as a response to the situation that caused the interrupt. Again we find that the content of the resulting streams ("wasted samples" and "actions") depends on time. Consequently, if we want to be able to use applicative languages in real-time programming, we must find a way to express this kind of time-dependence in a referentially transparent manner.

If we use the ambiguity operator *amb* of [McCarthy 63] we may express a solution to the above problem in an applicative syntax. Remember however that the use of amb destroys the referential transparency, since amb may not return the same result when applied repeatedly to the same arguments. We also generalize the solution so that both streams may show the same irregular behavior as the odds above. The function *clock t* is supposed to yield a time-out after *t* units of time. We also assume that we have operations that test if an object is an even or odd integer, or a timeout.

```
handle E O =
    letrec (Eh . Et) = E
      and (Oh . Ot) = O
      and next = amb Eh Oh
    in if even? next then
```

```
        let mate = amb Oh (clock t)
        in if timeout? mate then handle Et O
             else cons (add Eh Oh) (handle Et Ot)
      else if odd? next then
        let mate = amb Eh (clock t)
        in if timeout? mate then handle E Ot
             else cons (add Eh Oh) (handle Et Ot)
```

We may note that the original "Sum" program in Section 3 produced an ascending sequence consisting of every other odd integer. The time-sensitive program in this section still produces an ascending sequence of odd integers, although not every other one. The point is that certain properties of the original program are still preserved. Generally speaking, this is an important fact to note when we want to prove and verify our programs.

Another variation of this problem is the following: Suppose that we have an even integer, but no odd integer arrives before time-out. Then we want to throw away the next odd integer that arrives. If we have an odd integer, but no even, we want to get rid of the next even integer. To accomplish this, we would change the seventh line of our program to

```
if timeout? mate then handle E Ot
```

and the next to the last line to

```
if timeout? mate then handle Et O
```

In Section 3 we dicussed how we can cope with nondeterminism in an applicative language, using multisets and the constructor *frons*. The main point was that the thing that caused problems, namely nondeterministic behavior, was isolated into the data structure. Now our problem is that the computations we want to perform depend on

time. Could it be possible to embed time-dependent behavior in our data structures, in such a way that the resulting programs are referentially transparent?

Yes, it is! Assume that we have some objects called *hiatons* [Faustini 82], (from "hiatus", meaning a pause). A hiaton represents a unit of time. Assume furthermore that these hiatons can be elements of streams. That is, a stream could look like

(1 3 hiaton 5 hiaton hiaton 7 hiaton 9 11 .....

(From now on, we will occasionally abbreviate hiaton to H). Now suppose that the unit of time that elapses before a timeout, waiting for an odd integer in the original problem above, is the same amount of time as represented by a hiaton. Put another way, if the process that produces the odd integers haven't managed to produce an integer within this time limit, it outputs a hiaton instead. The stream output by this process will then look, for instance, like the stream shown a few lines above. Using streams with hiatons, we are now able to solve our problems, using a referentially transparent, applicative style of programming. In order to use an even integer only when a corresponding odd integer has arrived, and dispose of it otherwise, we write the following piece of code:

```
handle E O =
    let (Eh . Et) = E
      and (Oh . Ot) = O
    in if hiaton? Oh then cons hiaton (handle Et Ot)
         else cons (add Eh Oh) (handle Et Ot)
```

To show how this program works, let us suppose that the input streams begin as follows:

```
E = ( 2 4 6 8 10 12 14 16 18 20 .....
O = ( 1 H H 3  5  H  7  H  9 11 .....
```

The resulting output stream will then start with 3, since both 1 and 2 arrive before timeout. The next element of the even stream is 4, but in the odd stream we find a hiaton, so the result will be a hiaton. The even integer 4 is thrown away (or alternatively

output to a stream of wasted evens). The computation proceeds in this fashion, and produces the following result:

(3 H H 11 15 H 21 H 27 31 .....

To solve the other variant of the problem, to throw away the next odd if no odd arrives in time, we have to write a slightly more complicated function. We introduce an extra parameter to count how many of the next odd integers we are going to throw away.

```
handle E O count =
    let (Eh . Et) = E
      and (Oh . Ot) = O
    in if hiaton? Oh then cons hiaton (handle E Ot (add1 count))
      else if gt? nr 0 then handle E Ot (sub1 count)
      else cons (add Eh Oh)(handle Et Ot)
```

In this example we output a hiaton for each incoming hiaton in the odd stream. This is not the only possibility, but further discussion of this is deferred.

The fundamental difference between these later examples, and the previous examples is that in these functions the dependence upon time is embedded into the streams. If two streams both consist of (an initial segment of) the odd integers in ascending order, but the distribution of hiatons is different in the two streams, then we have got two different streams, although the numerical elements are the same , and in the same order. The expression *handle E O* will always give the same result when applied to the same arguments, because the phrase "the same arguments" now implies that the distribution in time of the stream elements is identical inasmuch as no hiaton occurs in one stream where an integer occurs in the other stream.

The introduction of hiatons does of course create some new questions. For example, should hiatons be introduced into all streams, connecting different processes? The

Accordingly, if we are to be able to use an applicative language for real-time programming, and especially in time critical tasks, we must be able to predict how long it will take to evaluate different expressions. In ordinary real-time languages this is done by compiling the statements in question into assembly code, and since we know the time required to execute each assembly instruction, it's not very difficult to compute the time needed for the corresponding high-level statements. As Wirth points out in [Wirth 77b], it wouldn't be too much to require that the compiler itself should provide us with accurate execution time bounds for any compiled statement or statement sequence.

However, a difficult problem occurs if our real-time system is programmed in an applicative language that uses suspended evaluation. As explained earlier, this means that expressions are evaluated when their values are needed, and that's often not when they first are encountered. This results in the possibility that if we need the value of an expression, this may cause the evaluation of certain other suspended expressions. But if we need the value of the first expression another time, the values of the previously suspended expressions are now computed, and our desired value can be returned almost immediately. The time required to evaluate our expression will not be the same on both occasions, and this conflicts with our need for predictable execution times.

One solution to this might be to make it possible for the programmer to declare which functions that are to be evaluated in lazy mode, and which expressions should be suspended. This approach is taken in [Moor 82] for the applicative language HOPE. This language is however primarily intended for sequential programming, and more research is needed to investigate if this approach can be combined with the need for suspended evaluation when using infinite streams and multisets.

Another problem is that applicative languages use recursion very heavily, in fact, using application recursively is the only "control structure". Although recursion permits us to express computations clearly and concisely, its implementation requires dynamic storage allocation and some kind of garbage collection. Also, the use of dynamic (infinite) data structures implies a need for garbage collection. As previously stated, garbage collection occurring at indeterminate times destroys the predictability of execution times. However, this problem is simplified by using real-time garbage collection, [Baker 78].

So far, we have said nothing about the hardware in our system, in particular we have not specified if we have one or many processors. Most real-time systems currently in use have a single processor. However, as Wirth points out in [Wirth 77b], processor sharing makes the determination of execution-time bounds much more difficult. The situation becomes even worse if processes are assigned different priorities, and processes with high priority may interrupt processes with lower priority. After a thorough analysis of the situation, Wirth concludes that the best way to solve the problem is to avoid it, that is, to dedicate one processor to each process. With rapidly decreasing hardware costs, this is certainly not as out of the question as it was ten years ago.

This conclusion is equally valid for an applicative real-time system, or maybe even more, since the opportunity to express and exploit parallellism is better in applicative languages than in current programming languages (see [Pettorossi 81]). The implementation of hiatons will probably also be less difficult if each process has its own, dedicated processor. The reader is advised to consult e.g [Musstopf 78] for further discussion of the impact of microprocessors on real-time systems.

5. Summary and Conclusions

We have been investigating the nature of real-time programming, and if applicative languages could be used for real-time programs. We found that the main problem was that certain real-time applications showed a nonfunctional, time-sensitive behavior. The rate of input determined not only the rate of output, but also what was output. However, by introducing a new kind of object called a hiaton we may solve this problem. A hiaton represents a unit of time, and codes up timing information. By allowing hiatons as elements of the streams used for communication between processes, we embed the dependence on time into the data structure, and may in this way preserve the functionality of all processes.

We have also seen that applicative languages meet most of the general requirements on real-time programming languages, e.g clarity, natural modularity, and ease of learning and use, and do also support parallel processing, process communication and synchronization. All this taken together shows that applicative languages would be well suited for real-time programming. Of course, a lot of research is still required to work out all the details, but the important point is that we have not found any fundamental, theoretical obstacle for using applicative languages in real-time systems.

One advantage with applicative languages is that their purely funcional, mathematical nature makes them easier to reason about than the imperative languages of today. Niklaus Wirth states in [Wirth 77b] that current real-time systems are, in practice, impossible to verify analytically. A real-time system, programmed in an applicative language, at least conceptually thought of as implemented on a multiprocessor system with one processor dedicated to each process, would open up new possibilities for verification. After verification, program transformation techniques may be applied to

transform the program into a more efficient and easy to implement program.

Acknowledgements. We would like to thank David Wise for his helpful comments and suggestions. We would also like to thank Sonja Edblom for her help during the preparation of this paper.

6. References

[Backus 78]
Backus J., Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Comm. ACM 21,* 8 (August 1978), 613-641.

[Baker 78]
Baker H. G. Jr., List Processing in Real Time on a Serial Computer. *Comm. ACM 21,* 4 (April 1978), 280-294.

[Barnes 74]
Barnes J.G.P., RTL/2 Language Design. In *[SPL 74a],* pp 6-9.

[Dennis 77]
Dennis J.B., A language design for structured concurrency. In *Design and Implementation of Programming Languages,* J.H. Williams, D.A. Fischer (eds.), Springer 1977, pp 231-242.

[Dijkstra 68]
Dijkstra E.W., Cooperating sequential processes. In *Programming Languages,* F.Genuys (ed), Academic Press, New York, 1968, pp 43-112.

[Dreisbach & Weissman 77]
Dreisbach T.A., Weissman L., Requirements for real-time languages. In *Design and Implementation of Programming Languages,* J.H Williams, D.A Fischer (eds.), Springer 1977, pp 298-312.

[Faustini 82]
Faustini A.A., *An Operational semantics for pure dataflow.* Dept. of Computer Science, University of Warwick, Coventry, UK, 1982.

[Friedman & Wise 79]
Friedman D.P, Wise D.S, An approach to fair applicative multiprogramming. In *Semantics of Concurrent Computation,* G. Kahn (ed.), Springer, 1979, pp 203-226.

[Friedman & Wise 80]
Friedman D.P., Wise D.S., An indeterminate constructor for applicative programming. In *Seventh ACM Symposium on Principles of Programming Languages,* 1980, pp 243-250.

[Heller et al 78]
Heller G., Kneis W., Rembold U., Weisner G., Standards and proposals of industrial real-time FORTRAN. In *[IFAC 78],* pp 95-107.

[Henderson & Morris 76]
Henderson P., Morris J.H.Jr., A lazy evaluator. *Third ACM Symposium on Principles of Programming Languages,* 1976, pp 95-103.

[Henderson 80]
Henderson P., *Functional Programming: Application and Implementation,* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[Henderson 82]
Henderson P., Purely functional operating systems. In *Functional Programming and its Applications,* J.Darlington, P.Henderson, D.A.Turner (eds.), Cambridge University Press 1982, pp 177-192.

[Hoare 74]
Hoare C.A.R., Monitors: An operating system structuring concept. *Comm. ACM 17,* 10 (Oct. 1974) 549-557.

[Holden & Wand]
Holden J., Wand I.C., Experience with the programming language MODULA. In *Real Time Programming 1977,* Proc. IFAC/IFIP Workshop, Eindhoven, Netherlands, 20-22 June 1977, pp 3-11.

[Ichbiah et al. 79]
Ichbiah J. et al., Rationale for the design of the Ada programming language. *Sigplan Notices 14,* 6B (June 1979), Chapter 11.

[IFAC 78]
IFAC/IFIP workshop on real time programming, Mariehamn, Finland, 19-21 June 1978. *Annual Review in Automatic Programmning, vol 9,* part 2/3.

[Infotech 71]
Real Time. International Computer State of the Art Report, Infotech, England, 1971.

[Keller et al. 79]
Keller R.M., Lindstrom G., Patil S., A loosely-coupled applicative multiprocessing system. *Proc. National Computer Conference 1979,* pp 613-622.

[Landin 65]
Landin P.J., A correspondence between ALGOL 60 and Church's lambda notation - part 1. *Comm ACM 8,* 2 (Feb. 1965), 89-101.

[McCarthy 63]
McCarthy J., A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems,* P. Braffort and D.Hirschberg (eds.), North-Holland, Amsterdam (1963), pp 33-70.

[Moor 82]
Moor I.W., An Applicative Compiler for a Parallel Machine. *Proc. of the Sigplan'82 Symposium on Compiler Construction, Sigplan Notices 17,* 6 (June 1982), pp 284-292.

[Musstopf 78]
Musstopf G. The influence of microprocessors on future real-time systems. In *[IFAC 78], pp 21-29.*

[Pettorossi 81]
Pettorossi A., An approach to communications and parallelism in applicative

languages. In *International Colloqium on Formalization of Programming Concepts*, J. Diaz, I. Ramos (eds.), Springer 1981, pp 432-446.

[Pike 71]
Pike H.E., Real Time Software for Industrial Control. In *[Infotech 71]*, pp 405-424.

[Schoeffler 71]
Schoeffler J.D., A real time language for on-line application and systems programming. In *[Infotech 71]*, pp 467-486.

[SPL 74a]
SPL Review; RTL/2 Special. SPL International, London, 1974.

[SPL 74b]
Introduction to RTL/2. SPL International, London, 1974.

[Wirth 77a]
Wirth N., Modula: A language for modular multiprogramming. *Software - Practice and Experience 7*, 1 (Jan. 1977), 3-35.

[Wirth 77b]
Wirth N., Toward a Discipline of Real-time Programming, *Comm. ACM 20*, 8 (August 1977), 577-583.