Information Processing Letters 30 (1989) 51–56
North-Holland

16 January 1989

REAL-TIME PROGRAMMING

Eric C.R. HEHNER

Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Canada M5S 1A4

Communicated by W.M. Turski
Received 29 January 1988

We try to say what constitues real-time programming by looking at some examples. We also comment on the kind of communication primitives that aid this programming, and the kind of formal semantics that is required.

Keywords: Real-time, communicating processes

Introduction

The term "real time" was first used in contrast to "simulated time"; by comparing the value of the "time" variable in a simulation with the real execution time of the simulation program, one obtains a measure of simulation speed. The term "real-time programming" is commonly used now as a synonym for "control programming"; it describes the programming of a device that must interact with an impatient physical environment. One is supposed to think of a chemical plant or nuclear plant where a device must react quickly to its inputs to control the chemical or nuclear process. Reacting to the keystrokes on the keyboard of a word processor presents the same sort of programming problem, although the penalty for slowness may be less consequential. It is hard to think of a device that does not have to react within a limited time.

One approach to this (or any) programming problem is to choose a powerful semantic formalism that allows us to express every sort of property that we suspect we may want to express. In particular, when time is a concern, we may choose temporal logic, or we may choose to base our logic on a clock with indexed ticks that allow us to refer to any instant. We shall take the opposite approach: we begin with a simple formalism that does not refer to time, and complicate it grudgingly when it becomes necessary.

Programming notations

We adopt some of the notations of communicating sequential processes. Output of the value of expression e on channel c will be denoted

$c!\, e$

Input is requested on channel c by the notation

$c?$

If input is not yet available on that channel, it is awaited. When the input is received, it is referred to simply as c. Our decision not to couple input with variable assignment simplifies the semantics, and it saves each programmer from introducing a variable for the purpose, whose appropriate name is the same as the channel name.

If c and d are input channels, and P and Q are programs (or program fragments), then

$c?\, P \,\square\, d?\, Q$

denotes input choice. It is executed as follows: if no input is available on either channel, input is awaited; the first input received determines which

one of the choices (either $c?P$ or $d?Q$) is executed; if input is already available on one channel but not on the other, the corresponding choice is executed; if input is already available on both channels, an arbitrary one of the choices is executed. Without adding time to the semantic formalism, we have no means to say "first". Fortunately, we are saved from having to add time by the following observation: an input choice must be designed to work no matter which input arrives first, hence no matter which input is chosen (except that it cannot wait forever for one input when the other is available). For logical correctness, there is no need to say "first". An implementation that chooses the first available input will be correct, and usually (but not always) the most efficient. We return to this point in the section titled "Time complexity".

We use $P;Q$ and $P \| Q$ for sequential and parallel composition of programs P and Q. We use $*P$ for the infinite repetition of program P. The precedence of these operators is: * before ; before $\square$ before $\|$ unless parentheses indicate otherwise.

Buffering

The channels used by a process and its environment to communicate with each other are called "external" to that process. A process may be composed of subprocesses communicating with each other; the channels they use are called "internal" to the process (but external to the subprocesses). When we introduce a new internal channel, we must decide the answers to the following questions.

(A) What is the buffer size? It can be zero, infinite, or anything between.

(B) When the buffer is full, does the sender wait? (We assume that when it is empty, the receiver waits.)

(C) When the buffer is full, if the sender does not wait, which message is lost? (Two possibilities are: the oldest or the newest.)

In CSP channels have buffer size zero (so they are always full), and the sender waits until the receiver is ready (and question (C) does not apply). For our internal channels we choose infinite buffers. These buffers are never full, so questions (B) and (C) do not apply. This choice is the simplest one for the semantics (as we see in the next paragraph), and is usually the most convenient for programmers. When it is not, we must create the sort of communication we want, as we shall see in the examples.

Here is an outline of the semantics of infinitely buffered communication. Each channel is a list-valued variable with an index. In the outputting process, the list is initially empty, and output is catenation to the end of the list. The final value of the list in the outputting process is taken as the initial value of the list in the inputting process. This does not mean that all output is produced before any input is read, but only that the input is the result of the output. Input simply advances the index in the list. This semantics suffices even when there are feedback loops in the communication. For further details, see [1].

An implementation cannot provide an infinite buffer. Nor can it provide the integer data type, nor the list data type, nor the operators to compose programs (sequential composition, conditionals, ...) in their full generality. It can provide only a finite approximation. In a language, it is as reasonable to have infinite buffers as it is to have integers or lists or program composition operators.

For external channels, we do not need to answer the three questions posed earlier. From a programmer's point of view, the answers are irrelevant. A program must direct a computer to function correctly for the data that it receives, regardless of whether the data were buffered, whether the sender waited, or which messages may have been discarded. Of course, "external" is a relative term; still, the questions are irrelevant whenever a process is considered by itself, rather than as a component of a larger program. Perhaps the point can be made more strongly: when we consider two communicating processes, we can ask whether communication between them is buffered, but when we consider just one of them, the question has no meaning.

Example: Keyboard

Consider now a computer with keyboard input and screen output. Its overall program might be divided into two processes communicating on an internal channel called buf.

*(keyboard?; buf! keyboard)

||*(buf?; process buf)

The front process receives input on the keyboard (an external channel), and communicates these inputs to the back process on the internal channel. The back process receives these inputs and processes them (generates appropriate responses on the screen). This division into processes with a buffer serves no logical purpose, but it has a physical purpose. We may want to dedicate a special processor and buffer to the keyboard, so our program structure matches the intended hardware structure. If there is only one processor, we want to ensure that the keyboard input gets enough processor time so that no characters are lost. (Having made the division, it is reasonable to let the front process do a few other things not shown here, such as compose characters from keystroke combinations and durations.)

In general, if one process feeds data to another process faster than the receiver can process it, a buffer will not solve the problem; it will fill up indefinitely, and the combination will run at the speed of the slower process. But in many cases, a buffer is helpful. In the keyboard example, there is a natural mechanism to ensure that the buffer does not fill up indefinitely: the human using the machine will want to see some output before continuing to provide input. The buffer smoothes this natural feedback so that the human can type as fast or as slow as desired.

Example: Reaction controller

We now turn to the problem of a reaction controller in a chemical plant. We suppose there is a sampler and digitizer that provides input at a certain rate. The controller must be fast enough to control the reaction properly; suppose it can do so by reacting to sufficiently many of the inputs, allowing some to be lost. We then divide our controller as before into a front end and a back end, with a communication channel described as follows: the buffer size is one, the sender (front end) does not wait, and the older message is lost. Since this is not the sort of communication provided by our internal channels, we must program it. Here is the outline.

*(sampler? □ request? reply! sampler)

||*(request! "ready"; reply?; process reply)

The front end repeatedly listens for input from the sampler and for data requests from the back end. The back end requests data when it is ready, receives the reply, processes it, and repeats its cycle. This program structure has both a logical and a physical purpose. The logical purpose is to provide a kind of communication that differs from the one provided by our communication primitives. The physical purpose is again to identify processes that must be executed with sufficient speed to match the external world; the front end must be ready for all inputs, and the back end must provide output quickly.

Example: Watch

Our final example is a watch. For simplicity, the display D is an integer modulo something. (Its presentation using mixed bases is not our present concern.) There are two buttons: M is the mode button, and A is the advance button. The mode button changes between time-mode and set-mode. In time-mode, the display cycles through its values, changing once each time unit. In set-mode, the display cycles through its values, changing once at each press of the advance button. (The display could be set in a more sophisticated way if we were to consider its presentation using mixed bases.)

The watch example mentions time explicitly, requiring us to represent if. The previous examples were not really concerned with time; their devices must be quick enough to match external events, however fast or slow that may be. But a watch

must tick at a particular speed. How can we program it to do so? A little reflection tells us that time is defined physically, not logically; it must be given, and cannot be specified. It can be the basis of a semantic formalism [3], or it can be provided as a sequence of ticks on a communication channel. Taking the latter suggestion, in time-mode the watch behaves according to the following program:

TimeMode:
(M? SetMode $\square$ tick? $D := D + 1$; TimeMode)

This program must be executed fast enough to receive each input as it occurs. If the M button is pressed, the watch goes into set-mode; if a tick is received, D is increased by one (modulo something), and the watch stays in time-mode. We define set-mode similarly.

SetMode:
(M? TimeMode $\square$ A? $D := D + 1$; SetMode)

Again, inputs must be received as they occur. The watch is started in either of its two modes.

So far, we have considered the ticks to be an external channel. Now let us consider the source of ticks (quartz) as a process in parallel with the rest of the watch. What happens to a tick while the watch is in set-mode? We do not want it to be buffered and processed later. We want the channel from the quartz to have the following characteristics: the buffer size is zero, the sender does not wait, and the message is lost if the receiver is not ready. (If we consider the human process, we can similarly say that pressing the advance button in time-mode is lost.) Since this is not the sort of communication we have chosen to provide, we must program it by modifying our definitions as follows.

TimeMode:
(M? SetMode $\square$ A? TimeMode
$\square$ tick? $D := D + 1$; TimeMode)
SetMode:
(M? TimeMode $\square$ A? $D := D + 1$;
SetMode $\square$ tick? SetMode)

The next refinement is a step towards implementation as a circuit: it introduces boolean variable m in a single loop.

$*$(M? $m := \neg m$
$\square$ A? **if** m **then** $D := D + 1$
$\square$ tick? **if** $\neg m$ **then** $D := D + 1$)

In all applications, a process must wait for any data it needs that are not yet ready, and it must produce its output fast enough for its purpose, whether that purpose is to control a chemical reaction or to inform an impatient human. Those are the constraints on its speed. It is rare that a process must produce its output at exactly a given time, rather than just "fast enough".

Time complexity

A semantics without time is simpler than one with, and it allows us to perform transformations and optimizations. For example, we can write

$$(x := x + 1;\ x := x + 2) = (x := x + 3)$$

if we are considering only the input-output relation and not the execution time. We would like to retain these benefits, but be able to calculate the execution time when it is needed.

We prove time properties the same way we prove other properties, using the same semantics. Let P be a program, and let t (for time) be a fresh variable (not appearing in P). Form program Pt from program P as follows. Begin with $t := 0$ setting the time to zero. Replace each assignment $x := e$ with the pair of assignments

$$(t := t + a;\ x := e)$$

where a is the time needed to evaluate e and perform the assignment. This information must be obtained from a knowledge of the implementation. Replace each output $c!e$ with

$$(t := t + b;\ c!e)$$

where b is the time needed to evaluate e and perform the output. Replace each conditional program

if e **then** Q **else** R

with

$(t := t + c;$ **if** e **then** Q **else** $R)$

where c is the time needed to evaluate e and perform a branch. The loop construct and sequential composition do not need any changes. We will consider parallel composition, input, and input choice in a moment. Then to prove something about the execution time of P is just to prove something about the final value of t in Pt. In general, it will depend on the initial values of the other variables.

In a parallel composition, we introduce a temporary time variable for each process, and then take the maximum. Replace $Q \| R$ with

$(tq := t; tr := t; (Qt \| Rt); t := \max\ tq\, tr)$

where tq and tr are fresh, Qt is formed from Q using time variable tq, and Rt is formed from R using time variable tr.

An input request $c?$ from an external channel must be replaced by

$(t := t + w; c?)$

where w is the length of time spent waiting for the input. This time may not be knowable, so an assumption must be made. It is reasonable to assume that w is zero, that inputs are always ready, and to affix a note to any conclusions saying that times will be increased by any delays waiting for input. In some circumstances, the time that an input is ready may be known. If an input is ready at time r, then its input request $c?$ must be replaced by

$(t := \max\ tr; c?)$

Input choice can be treated simply by treating its inputs as above (for syntactic reasons, the time change must be placed after the input, rather than before). However, if we know that the implementation always chooses the first available input, we can do better. The input choice $c?Q \square d?R$ becomes

$\big(t := t + (\min\ wc\ wd); \big(c?Q \square d?R\big)\big)$

where wc is the waiting time for channel c and wd is the waiting time for channel d. Alternatively, if rc and rd are the times that the communications will be ready, we can use

$\big(t := \max\ t\ (\min\ rc\, rd); \big(c?Q \square d?R\big)\big)$

Internal communication channels between parallel processes present a problem. Here is an example in which the spaces represent waiting for time-consuming evaluations of the two expressions $e6$ and $e8$.

```
x := e6; c! x;          d?; z := d
||          c?; Y := e8; d! y
```

The time spent waiting for input is not an unknown, but it depends on another process. In more complicated examples, the dependencies can be complicated. Recall that parallel processes Q and R give us separate time variables tq and tr. Suppose channel c goes from Q to R. Within Q replace $c!\ e$ with $(tq := tq + b;\ c!\ tq;\ c!\ e)$ where b is the time needed to evaluate e and perform the output. Within R replace $c?$ with $(c?;\ tr := \max\ tr\ c;\ c?)$. Suppose assignment and output each take time 1, evaluation of $e6$ takes time 6, and evaluation of $e8$ takes time 8. Then the above example becomes

```
(tq := t: tr := t;
(   (tq := tq + 7; x := e6);
    (tq := tq + 1; c! tq; c! x);
    (d?; tq := max tq d; d?);
    (tq := tq + 1; z := d)
||  (c?; tr := max tr c; c?);
    (tr := tr + 9; y := e8);
    (tr := tr + 1; d! tr; d! y))
t := max tq tr)
```

The time gets passed back and forth between the processes. (If the type of channel value is not numerical, a separate channel may be defined for the purpose.)

Calculating the execution time of a program using the calculus we have presented is a formidable task; it has all the problems of calculating the semantics. Although it may never be practical to calculate the exact time, it is practical to calcu-

late upper bounds: whenever an expression becomes unwieldy, it can be replaced by a greater but simpler one. By analogy with semantics calculation, we suppose the calculus is much more useful in the construction of programs whose execution time bounds are given than in the calculation of execution time bounds of given programs.

Conclusion

If a device does not react fast enough to its inputs, there are three options: to obtain faster hardware (perhaps more processors in parallel), to write a better program, or to concede that we cannot currently automate the process. It is of no use to invent a more powerful semantic formalism (e.g. one based on time). To know whether a device will react fast enough, we need a timing calculus; once again, this does not require us to complicate the semantic formalism.

"Real-time programming" is a term in search of a definition; I am not very concerned whether it finds one. The examples show some common characteristics: a separation of the task into processes whose execution speeds must match particular external input or output speeds. The input choice operator is used to merge input streams. Perhaps real-time programming is just this programming paradigm.

Acknowledgements

This paper was inspired by a talk given by Wlad Turski at IFIP W.G.2.3. The watch example was suggested by N. Halbwachs. The method of calculating time complexity for sequential programs comes from [2] (except that Lengauer used the "weakest precondition" calculus and consequently his clocks ran backwards). We have extended the method to parallelism and communications.

References

[1] E.C.R. Hehner, Predicative programming, *Comm. ACM* **27** (2) (1984) 134–151.

[2] C. Lengauer, A methodology for programming with concurrency; the formalism, *Sci. Comput. Programming* **2** (1982) 19–52.

[3] J.-L. Bergerand, P. Caspi, N. Halbwachs and J.A. Plaice, Automated control systems programming using a real time declarative language, In: *4th IFAC/IFIP Symposium on Software for Computer Control (SOCOCO)*, Graz Austria, 1986.