# Two Exercises with Real-Time Programs

## 0 Introduction

A real-time program is a (parallel) program whose correctness depends on assumptions about the relative speeds of execution of its components. In this note we try to develop suitable notations for recording such speed assumptions in the text of the program. Thus, these assumptions become *requirements* imposed upon the implementation of the program: the program is correctly executed provided the machine meets these requirements. This brings about the need for a (so-called) real-time scheduler, but here we are not concerned with the construction of schedulers; in particular we shall not address the relative merits of compile-time scheduling versus run-time scheduling. Obviously so, because we must, of course, discover the relevant concepts first, and formulate them in an implementation independent way, and only then can we address the (important) issue of their implementation.

In this note we shall use *guarded statements* as synchronisation primitive. My first experiments leave me with the impression that the following discussion is not very sensitive to what synchronisation primitives are used, as long as they are decent[0], and guarded statements are conceptually the simplest.

A guarded statement has the shape $\mathsf{if}\ B \rightarrow S\ \mathsf{fi}$, where $B$ is a boolean expression and $S$ is a statement. Its operational meaning is that $S$ can only be executed in states where $B$ —the guard— has the value $\mathsf{true}$; as long as the guard is $\mathsf{false}$ the process executing this statement is *blocked*. Formally we have that $\{\ P\ \}\ \mathsf{if}\ B \rightarrow S\ \mathsf{fi}\ \{\ Q\ \}$ is equivalent to $\{\ P \wedge B\ \}\ S\ \{\ Q\ \}$.

As an abbreviation we shall use $[B]$ instead of $\mathsf{if}\ B \rightarrow \mathsf{skip}\ \mathsf{fi}$, and we use $*[\ S\ ]$ to denote the eternal repetition of statement $S$.

## 1 First example

We consider a parallel program consisting of two components named $X$ and $Y$, in which $S_0$ and $S_1$ are statements in which variables $x$ and $y$ do not occur; $x$ and $y$ are auxiliary variables only, to express the required synchronisation of the two components:

[0] It cannot be said too often: the "event-mechanism" is not decent.

precondition : $x = y$
invariant : $y \leq x \ \wedge \ x \leq y+1$

component $X$ : $*[\ \{\ x \leq y\ \}\ x := x+1$
$;\ S_0$
$;\ [x \leq y]$
$]$

component $Y$ : $*[\ [y < x]$
$;\ S_1$
$;\ y := y+1$
$]$

If so desired, $x := x+1$ can be interpreted as "send a signal to process $Y$" and $[y<x]$ can be interpreted as "wait for a signal from process $X$"; if initially $x=0$ the value of $x$ then represents the number of signals sent from $X$ to $Y$.

The invariant of this program consists of the two conjuncts $y \leq x$ and $x \leq y+1$. In the development of the above program these two conjuncts play entirely separate roles: the one conjunct, namely $y \leq x$, gives rise to the guarded statement $[y<x]$, whereas the other one, $x \leq y+1$, gives rise to the other guarded statement $[x \leq y]$.

The purpose of the statement $[x \leq y]$ is to establish the local correctness of the assertion $x \leq y$, which is the precondition of $x := x+1$. If we are able to establish the correctness of this assertion by *other* means we may safely omit the statement $[x \leq y]$. The problem to be solved now is that we must eliminate this statement because we *assume* that, for reasons irrelevant here, component $X$ may not contain guarded statements.

* * *

As stated above, this problem involves no notion of *time*, real or otherwise. If component $X$, however, may not contain guarded statements then all we can do to guarantee the invariance of $x \leq y+1$ is to introduce additional assumptions about the relative speeds of the two components. Roughly speaking, our problem is solved if we can see to it that component $Y$ is sufficiently *fast* with respect to $X$, or, equivalently, that $X$ is sufficiently *slow* with respect to $Y$.

We formalise this as follows. First, we introduce a fresh variable $t$ and an additional component $Z$, which can be considered as an (abstract) "clock":

component $Z$ : $*[\ t := t+1\ ]$

Second, we introduce a fresh variable $p$ and we modify component $X$ as follows:

component $X$ : $*[\ x, p := x+1, t$
$;\{\ p \leq t\ \}$
$S_0$
$;\{\ p+T<t$ (see below) $\}$
$[x \leq y]$
$]$

Here we take the local correctness of the assertion $p+T<t$ for granted: we *assume* that this is a property of $S_0$, that is, we assume that $S_0$ satisfies $\{\ p \leq t\ \}\ S_0\ \{\ p+T<t\ \}$. This specifies formally that $S_0$ is "slow enough": its execution takes more than $T$ ticks of our clock.

Now we may indeed omit the statement $[x \leq y]$, provided that it admits as additional precondition:

$$p+T<t\ \Rightarrow\ x \leq y\ ,$$

which can be rewritten into an equivalent but more convenient form as:

$Q$: $\quad t \leq p+T\ \vee\ x \leq y\ .$

So, we add $Q$ as an additional assertion to component $X$, as follows; its local correctness follows from the additional (but quite reasonable) assumption $0 \leq T$:

component $X$ : $*[\ x, p := x+1, t$
$;\{\ p \leq t\ \}\{\ Q\ \}$
$S_0$
$\{\ p+T<t \wedge Q,$ hence: $\}$
$\{\ x \leq y\ \}$
$]$

As for $Q$'s global correctness we observe that the only conflicting statement is $t := t+1$ from component $Z$. Fortunately, the statement $y := y+1$ in component $Y$ establishes $x \leq y$ and this implies $Q$; we can now prevent violation of $Q$ by $t := t+1$ by seeing to it that $y := y+1$ is executed "on time", that is, before $t$ exceeds $p+T$. More precisely, $y := y+1$ must be executed (and completed!) in a state where $t \leq p+T$ (still) holds. (This is operational reasoning but, alas, currently I can do no better.)

This is not a conventional synchronisation requirement and to express it we need some new notation. We shall use $\mathsf{if}\ S \leftarrow B\ \mathsf{fi}$ to state that the execution of statement $S$ must *terminate before* $B$ *will unbecome* $\mathsf{true}$. In a way this construct is the dual —under reversal of the direction of time— of $\mathsf{if}\ B \rightarrow S\ \mathsf{fi}$: the latter can be explained by the requirement that execution of $S$ must *begin after* $B$ *has become* $\mathsf{true}$.

We apply this new construct in component $Y$. To obtain a better decoupling of $X$ and $Y$ we introduce a local variable $q$ that will satisfy $q{=}p$. Strictly speaking, $q$ is superfluous, but by means of it we can formulate the "timing requirement" for $Y$ in more local terms; the place of the assignment $q{:=}p$ requires justification, which I will try to give later:

$$
\begin{array}{ll}
\text{component } Y: & *[\ \mathsf{if}\ y{<}x \rightarrow q{:=}p\ \mathsf{fi} \\
& ;\{\ y{<}x \wedge q{=}p\ \} \\
& \quad S_1 \\
& ;\ \mathsf{if}\ y{:=}y{+}1 \leftarrow t{\leq}q{+}T\ \mathsf{fi} \\
& ]
\end{array}
$$

We might wish to leave implicit the explicit clock variable $t$ and variable $p$ (which more or less belongs to component $X$) and so we might introduce abbreviations for the two (relevant) statements in $Y$, for example:

$$
\begin{array}{ll}
\text{component } Y: & *[\ [y{<}x\langle ?q\rangle\,]\ \\
& ;\{\ y{<}x \wedge q{=}p\ \} \\
& \quad S_1 \\
& ;\ y{:=}y{+}1\,\langle{\leq}q{+}T\rangle \\
& ]
\end{array}
$$

In this way we obtain something that resembles the timing annotations used by Onno van Roosmalen. The above derivation shows how to interpret these timing annotations properly; in particular, $[y{<}x\langle ?q\rangle\,]$ means "assign to $q$ the value of the clock at the moment $y{<}x$ becomes $\mathsf{true}$"; phrased differently, $q$ represents the state of the clock at the moment a signal is sent, not the moment it is received. Similarly, $y{:=}y{+}1\,\langle{\leq}q{+}T\rangle$ can be interpreted as "complete $y{:=}y{+}1$ before the clock strikes $q{+}T$".

* * *

We now analyse what freedom we have in placing the assignment $q{:=}p$ in component $Y$. Initially, I combined it immediately with the guarded statement $[y{<}x]$, inspired by the way in which Onno includes timing annotations in his programs. This is somewhat strange, however: the timing requirement

is introduced for the sake of the invariance of $x \leq y+1$, whereas the statement $[y<x]$ serves the invariance of $y \leq x$ and, as we have seen, these two parts of the invariant are completely independent. Nevertheless, we really have no choice, for the following reasons.

First, the correctness of the program crucially depends on the assumed slowness of $S_0$; by means of the statement $x, p := x+1, t$ the moment at which $X$ is about to perform $S_0$ is recorded in $p$ and, via $q$, communicated to $Y$. Second, in order to complete $y := y+1$ on time, component $Y$ must complete $S_1$ first, that is, $Y$ must complete $S_1 ; y := y+1$ on time. Now it seems reasonable to require that, for the sake of implementability, $q := p$ precedes $S_1$. Finally, the global correctness of the assertion $q = p$ depends on the disjointness of the conjunct $y<x$ and the precondition $x \leq y$ of $x, p := x+1, t$ [1]. Therefore, $q := p$ must have $y<x$ as its precondition. Hence, the only place where we can insert $q := p$ is *after* $[y<x]$ and *before* $S_1$. (Notice that, because $y<x$ is *stable* the statement $\mathsf{if}\ y<x \rightarrow q := p\ \mathsf{fi}$ need not be atomic.)

We may interpret $x, p := x+1, t$ as "send the value of $t$ to $Y$" and we may interpret $\mathsf{if}\ y<x \rightarrow q := p\ \mathsf{fi}$ as "receive a value from $X$ and assign it to $q$". This communication is asynchronous and this is fine, under the proviso that time spent in this communication is not available anymore for performing $S_1 ; y := y+1$: the longer the communication takes, the stronger the timing requirement for $S_1 ; y := y+1$ becomes.

In practice we probably prefer to use explicit communication statements instead of shared variables and guarded statements. This is easy; we simply rewrite our program into the following (and for the time being final) form. Now $p$ and $q$ are local variables of components $X$ and $Y$, and now $\langle ?p \rangle$ just means $p := t$, that is, "assign the value of the clock to $p$":

component $X$ : $\quad *[\ \langle ?p \rangle\ ;\ \mathit{send}{\cdot}p\ ;\ S_0\ ]$

component $Y$ : $\quad *[\ \mathit{receive}{\cdot}q\ ;\ S_1 \langle \leq q+T \rangle\ ]$

## 2 Second example

As a second example we now consider a parallel program, for the implementation of $*[\ S_0 \,||\, S_1\ ]$, again consisting of two components $X$ and $Y$:

[1] This is an application of what nowadays is called the "rule of disjointness".

precondition : $x = y$
invariant : $y \leq x+1 \ \wedge \ x \leq y+1$

component $X$ : $*[ \ [x \leq y]$
$\ ; \ S_0$
$\ ; \ x := x+1$
$\ ]$

component $Y$ : $*[ \ [y \leq x]$
$\ ; \ S_1$
$\ ; \ y := y+1$
$\ ]$

As before, we assume that component $X$ may not contain guarded statements; therefore, we have to eliminate the statement $[x \leq y]$ from $X$. We apply the same technique and modify $X$ and $Y$ as follows:

assertion $Q$ : $t \leq p+T \ \vee \ x \leq y$

component $X$ : $*[ \ \{ \ x \leq y \ \} \ p := t$
$\ ; \{ \ p \leq t \wedge Q \ \}$
$\quad S_0$
$\ ; \ x := x+1$
$\quad \{ \ p+T < t \wedge Q, \text{ hence: } \}$
$\quad \{ \ x \leq y \ \}$
$\ ]$

component $Y$ : $*[ \ \mathsf{if} \ y \leq x \rightarrow q := p \ \mathsf{fi}$
$\ ; \{ \ y \leq x \wedge q = p \ (?) \ \}$
$\quad S_1$
$\ ; \ \mathsf{if} \ y := y+1 \leftarrow t \leq q+T \ \mathsf{fi}$
$\ ]$

Unfortunately, we cannot establish the correctness of the assertion $q = p$ in component $Y$: the preconditions of $q := p$ and $p := t$ are not disjoint. The only thing we can do is introduce an additional variable $z$ and use it to obtain disjoint assertions:

precondition : $x = z$
invariant : $z \leq x+1$

component $X$ : $*[\ \{\ x \leq y \wedge x = z\ \}\ z, p := z+1, t$
$;\ \{\ p \leq t \wedge Q\ \}$
$S_0$
$;\ x := x+1$
$\{\ p+T < t \wedge Q,$ hence: $\}$
$\{\ x \leq y\ \}$
$]$

component $Y$ : $*[\ [y \leq x]$
$;$ if $y < z \rightarrow q := p$ fi
$;\ \{\ y < z \wedge q = p\ \}$
$S_1$
$;$ if $y := y+1 \leftarrow t \leq q+T$ fi
$]$

So, the communication of $p$ requires an additional synchronisation: component $Y$ cannot receive a value before it has been sent. In this case this additional synchronisation can take over the role of the old statement $[y \leq x]$: because $[y < z \wedge z \leq x+1 \ \Rightarrow\ y \leq x]$ we have that $y \leq x$ is a valid postcondition of the new guarded statement. Thus, we obtain the following program; notice that $x$ can be eliminated as its role is taken over by $z$:

precondition : $x = z$
invariant : $z \leq x+1$

component $X$ : $*[\ \{\ x \leq y \wedge x = z\ \}\ z, p := z+1, t$
$;\ \{\ p \leq t \wedge Q\ \}$
$S_0$
$;\ x := x+1$
$\{\ p+T < t \wedge Q,$ hence: $\}$
$\{\ x \leq y\ \}$
$]$

component $Y$ : $*[$ if $y < z \rightarrow q := p$ fi
$;\ \{\ y < z \wedge q = p \wedge y \leq x\ \}$
$S_1$
$;$ if $y := y+1 \leftarrow t \leq q+T$ fi
$]$

Hence, the reception of the value of $p$ and the synchronisation $[y \leq x]$ coincide. As before, we can now encode the program in terms of explicit communication statements; this happens to yield exactly the same program as in the previous example:

component $X$ : $*[\ \langle ?p \rangle\ ;\ \mathit{send}{\cdot}p\ ;\ S_0\ ]$

component $Y$ : $*[\ \mathit{receive}{\cdot}q\ ;\ S_1\ \langle \leq q{+}T \rangle\ ]$

## 3 Epilogue

When I set out to write this note I decided to include the second example because I wished to demonstrate that communications of clock values need not coincide with the synchronisations already present in the program. In both examples it so happens that these communications can be made to coincide with synchronisations already present. This is nice, of course, but it remains to be seen whether this will always be possible.

It does not come as a surprise (to me, at least) that in the two examples the state of the clock must be recorded in component $X$ and that, as a result, the time needed to communicate this value to $Y$ is relevant. When $S_1$ must be completed on time then everything preceding $S_1$ must be completed on time as well; in our case the preceding action is the communication. In particular, when $X$ is an "external" process and when the communication involves an *interrupt*, then the (maximal) amount of time elapsing before the machine responds to this interrupt is relevant, because this determines how much time is left to execute $S_1$.

To simplify the implementation it is always allowed to replace values by pessimistic estimates thereof. For example, if we know that the sequence $\langle ?p \rangle ; \mathit{send}{\cdot}p ; \mathit{receive}{\cdot}q$ takes at most $U$ ticks of the clock then we can formalise this knowledge by postulating the validity of $\mathit{receive}{\cdot}q\,\langle ?r \rangle\ \{\ r \leq q{+}U\ \}$. Because $t \leq r{-}U{+}T$ implies $t \leq q{+}T$ if $r \leq q{+}U$, the following version of our program is also correct. Variables $p$ and $q$ are eliminated as they are now superfluous and the (now value-less) *send* and *receive* reduce to semaphore operations. The timing annotation is now confined entirely to component $Y$, which may be attractive:

component $X$ : $*[\ \mathit{send}\ ;\ S_0\ ]$

component $Y$ : $*[\ \mathit{receive}\langle ?r \rangle\ ;\ S_1\ \langle \leq r{-}U{+}T \rangle\ ]$

As the examples in this note show, synchronisation requirements often consist of parts that can be treated in isolation. Some of these parts will be implemented by means of timing requirements, whereas other parts will be implemented by means of "ordinary" synchronisation primitives. Hence, besides timing annotations we shall always need decent ordinary synchronisation primitives. It goes without saying that the latter should not be blurred by the former.

Finally, I must confess that I have strong doubts about the feasibility of compile-time scheduling in those cases where the arrival times of signals from external processes display large variations, but that is quite a different story.

Eindhoven, 9 january 1995

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven