TIMING CONSIDERATIONS WILL DAMAGE YOUR PROGRAMS
HOW TO COPE WITH MANY PROCESSORS IN NO TIME AT ALL

W M TURSKI

Rapporteur:	**First Lecture**	M Pietkiewicz-Koutny
	Second Lecture	M Koutny

TIMING CONSIDERATIONS WILL DAMAGE YOUR PROGRAMS
OR
HOW TO TRADE CLOCKS FOR CHIPS

W.M.Turski
Institute of Informatics
Warsaw University

The traditional view of computation has been that of input/output transformation. Hence the abstract paradigm, variously represented as Turing machine, Hoare triple, Dijkstra predicate transformer, function application etc. Hence also the classical problem of program execution termination, the root of distinction between partial and total correctness, apparent unidirectionality of both control and data flows. As long as the act of computation is performed by a single processor, this view is perfectly adequate; it also proved very productive in terms of theories, programming language design and programming practice. Above all, it led to sound methodological developments, which - contrary to widespread lamentations - have resulted in vast improvements in most aspects of software over not quite half century of electronic information processing.

However, the traditional view suffers from two intrinsic limitations: it cannot incorporate any notion of time and it does not extend to essentially multiprocessor environments. Actually, it may be argued that these two limitations are in fact very closely related. Indeed, no physical notion of time can apply in a system that has fewer than two agents capable of autonomous (sponte suum) action. A sole agent experiences no measurable time; the question whether an existential, philosophical notion of time applies in such a case is best left outside the scope of these notes.

A program can be proven correct with respect to its specification only insofar as both program and specification are considered as formal objects. There is no way in which the 'real' time (astronomical epochs or durations) can be formalized. Logical calculi are impervious to notions of time: various temporal logics are merely about various orderings on discrete events. Thus we cannot prove any timing properties of a program, at least not in the sense in which we can prove its formally expressible properties. We can prove that a program execution eventually terminates, or even that it terminates after at most so many cycles. We cannot prove that 'eventually' is before next Friday noon. If we have a reasonably good complexity analysis of a program and of all supporting layers of software we may estimate the number of machine cycles needed to execute it. If we know - experimentally, or on trust - the duration of a machine cycle, we may estimate the running time of this program. But an estimate depending on an experiment and/or trust is not a proof.

Any knowledge about real time properties we may possess is not about programs but about programs implemented on a computer. As such, it is essentially experimental and depends on the proper-

ties of the machinery - it certainly does not allow any degree of portability. Indeed, as we cannot include in our programs any absolute tests for time epochs or intervals, we cannot even program in a time-safe way similarly as we condition our software to abort if the machinery does not support sufficiently large integers, sufficiently many different addresses, or performs arithmetic operations in an erratic way.

Thus, whenever we make any assumptions about any timing properties of the software, we are on intrinsically more dangerous grounds than in any other aspect of our risky profession. In fact, the scope for catastrophic errors of judgement in time-sensitive software is so large and fundamentally irreducible that any inclusion of time-related features should be simply avoided.

Can we do so? I am convinced that yes, we can avoid any considerations of time in programming. (Of course, this is not to say that we should avoid concerns of efficiency: heap-sort program is provably more efficient than naive-sort program, and hence 'faster' in a very large class of implementations.)

In these notes, I shall attempt to illustrate three methods of avoidng time-related considerations in program-design:

1. By undoing unnecessary simplifications.
2. By concentrating on local tasks.
3. By not worrying about the next step.

A lot of timing considerations are thrust on program designers by the existing practices in the application domains, chiefly (but not exclusively) in control engineering. Quite typically, an application problem can be clearly stated in its natural variables (such as pressure, temperature, chemical composition etc.)

- given that Q(x,y,...,z)
- by means of available transformations

  f(x,y,...,z),
  g(x,y,...,z),
  ...
  h(x,y,...,z)

- achieve R(x,y,...,z)
- maintaining P(x,y,...,z)

This is, of course, a perfect starting point for the design of a program. Very often, however, the problem is presented as already 'simplified' by resolving the relationships between the natural variables via equations in an independent variable - time. In general, the the simplification aim is to determine 'controls'

c1(t,x0,y0,...z0),
c2(t,x0,y0,...z0),
...
cN(t,x0,y0,...z0)

and a constant T, such that application of controls along the trajectory starting at time t=t0 in a state (x0,y0,...z0) that satisfies condition Q(x0,y0,...z0) will result at time t=T in a state satisfying condition R. Also in general, this goal turns out too difficult to achieve. A number of further simplifications are then introduced; very often the controls need to be approximated by piece-wise continuous functions, so that they are valid for a small time interval only, or in small neighbourhoods of (x0,y0,...z0) only, or both. Consequently, they have to be recomputed every so often, before they lose their current validity. A control program must be fast enough to do the control computations before the controlled process leaves a region. In addition, because the process development as a function of time is known only approximately, the size of the current neighbourhoods is not very well known, thus - to be on the safe side - it is replaced by an estimate of the time interval. We end up with the need to compute the controls within n seconds and nevertheless watch out for critical variables straying out of safety bounds. Of course, any implementation of such a control system is intrinsically impossible to prove correct. Often it is also unsatisfactory in terms of the original problem statement.

The added rationale of the simplification supposedly rests in the assumption that monitoring the natural variables is difficult (expensive) and slow, whereas monitoring time is easy and fast. (This is the rationale behind the simplification of the problem 'to cook a soft-boiled egg' to the problem 'to boil an egg for three minutes'.) In many cases the progress in metrology has levelled off any difference between monitoring natural variables and time, but the bias remains. Often, as with the three-minute egg, the original problem - totally time-independent! - is now presented as if its *central issue* was time. Well, if you look into the problem, not into its control-engineering simplification, you will probably discover it is not the case.

Another class of timing concers are entirely self-inflicted. Basically, they are a consequence of the belief that local frugality results in global optimality. A typical example of this class is the notorious timeout concept.

Usually, the timeout is presented as a preferred alternative to a potentially infinite busy-wait loop, the latter being 'obviously' wasteful and therefore to be avoided. The point which does not get a fair consideration is that the busy-wait is wasteful *only if there are better (useful?) things to do.* This being the case, it is not very logical to postpone these other things even for a short while, for the duration determined by the timeout constant (incidentally, the actual choice of this constant is frequently purely arbitrary). Thus, it would seem that the only justified situation for a timeouted wait is when there is nothing else left to do, but, under such circumstances, the potentially infinite busy-wait can hardly be considered as wasteful!

To be a little more specific let us consider two designs:

```
PR1: do
       WORK_TO_DO -> work
                  []
       EVENT_PRESENT -> react
                  []
       TRUE -> skip
     od
```

and

```
PR2: t:=0;
     while t<tmax and ~EVENT_PRESENT
       do t:=t+1
       if EVENT_PRESENT then react fi;
     work1
```

where tmax represents the timeout constant and procedure work1 may differ from procedure work because the former is executed in PR2 under the condition 'either the reaction to event has already taken place or the event will never be registered'.

PR2 is a fairly typical design with a timeout, PR1 is the busy-wait alternative expressed in a mildly nondeterministic fashion. It is quite apparent that the design PR1 corresponds to the natural idea of 'reacting to an event that may, but need not, occur' rather well, whereas the design PR2 introduces a curious 'window of opportunity' for the event to occur, which in many cases is hard to justify.

If the guard WORK_TO_DO in PR1 seems too coarse, a version with a partition of the corresponding subset of the state-space may be more appealing:

```
PR1':do
       WORK1_TO_DO -> work1
                   []
       WORK2_TO_DO -> work2
                   []
       ...

       WORKN_TO_DO -> workN
                   []
       EVENT_PRESENT -> react
                   []
       TRUE -> skip
     od
```

PR1' presents the natural way in which a modicum of nondeterminism helps to cope with problems of interaction with events not controlled by the program, without introducing an artifact of timeout. Assuming a fair implementation of PR1 type programs, the price - an endless waiting loop - is irrelevant because looping uses resources that could not be otherwise employed!

Observe that in PR2 we rely on Dijkstra semantics [DIJ] of guarded commands to resolve the nondeterminism of not-disjoint guards (a must for an implementation in a single-processor environment in view of the last guard: TRUE overlaps with any other guard!). In the final part of these notes we shall demonstrate how an essential use of a multiprocessor implementation eases this requirement. Observe also that having thrown away concerns with the mythical global efficiency, we do obtain a more natural solution which in fact is more efficient: if the event occurs while there is still some (independent of it) work left, no appreciable resources are wasted on the implementation of timeout.

A similar reasoning applies to the design (programming) of interacting processes, i.e. to the protocols. Assume we are concerned with two processes, P1 and P2, which do not share any variables (if they do, the interaction can be easily handled by mutual exclusion on access to these variables, certainly without any reference to time-considerations).

Timeouts are usually introduced by the designs which slavishly follow old-fashioned principles of telephone communication: When I call my friend, I dial, hear the ringing tone, and wait for my friend to pick the handset. I have to determine when to put my handset down if my friend does not answer: after 5 rings or after 10. Then I continue with my other work, perhaps calling my friend again when the conditions are once more deemed propitious. Of course, busy executives have long ago rejected this policy. They ask their secretaries to establish connection and continue with their work. When (and if) the connection is established, the secretary announces this fact, and if the executive is free to talk the conversation takes place. No timeout (but two 'processors')! Once again, only if no other work remains to be done without establishing the connection the calling executive waits for the call to go through. But, as before, under these (exceptional) circumstances an endless waiting loop is not wasteful: there is nothing to waste!

Denoting the variables of Pi by varPi (sets varP1 and varP2 are disjoint), and by MS(varP1) the predicate that is satsified iff P1 is ready to send a message to P2, we can adapt the design PR1' to our present problem

```
PRC: do
       WORK1_TO_DO -> work1
                   []
       WORK2_TO_DO -> work2
                   []
       ...
       WORKN_TO_DO -> workN
                   []
       MS(varP1)   -> send_message
                   []
       TRUE -> skip
     od
```

Note that PRC may send multiple copies of the message; if we want to make sure that this does not happen, a slightly modified version should be used

```
PRC': do
        WORK1_TO_DO -> work1
                    []
        WORK2_TO_DO -> work2
                    []
        ...
        WORKN_TO_DO -> workN
                    []
        MS(varP1)   -> send_message; make_MS_false
                    []
        TRUE -> skip
      od
```

How to reflect our concern with the reception (or otherwise) of the message by P2? The state space of P2 is inaccessible to P1, therefore we must assume that there is a predicate MR(varP1) - on variables of P1 - satisfied iff the message has been received by P2. With this predicate, we may design P1 as follows

```
PRC': do
        WORK1_TO_DO -> work1
                    []
        WORK2_TO_DO -> work2
                    []
        ...
        WORKN_TO_DO -> workN
                    []
        MS(varP1)   -> send_message
                    []
        MR(varP1)   -> set_MS_to_false
                    []
        TRUE -> skip
      od
```

This design ensures that no further copies of the message are sent out after P1 has been notified about the reception by P2; however, multiple copies may have been dispatched before, it is therefore up to P2 to disregard second and further copies of the same message. In practice, both MS and MR could appear as conjuncts in some (presumably, different) guards WORKi_TO_DO, WORKj_TO_DO; sending the message and setting MS to false would then constitute parts of corresponding actions worki, workj. In PRC' there is no explicit 'receive' command, even though a receiving action is necessary, if only to accept the acknowledgement signal(s) from P2. (An appearence of the acknowledgement signal could be treated as an event in the same sense in which this word was used before, and treated correspondingly).

The above outlined approach may seem wasteful insofar as it seems to encourage possibly unnecessary duplication of message sending actions (in addition to infinite waiting loops!). It should be however observed that, as far as process P1 is concerned, these 'unproductive' actions take place only if there is no other demand on resources. (Similar observation applies to P2.) The only 'overworked' element may be the network. Should this be a reason to worry, the network could be considered as an active partner, exercising full control on its resources, as long as there is useful work for it, and lapsing into wasteful actions otherwise.

The alternatives to timeout thus considered (cf. also [TUR]) suggest a much less sequential programming style: one in which specific actions occur under specific conditions, but nothing is explicitely said about sequences of thus guarded actions. Now we are going to extend this style of programming to an essentially multiprocessor environment.

Consider a set of variables, V, and for each its element, v, a set of possible values, Valv. A state is any mapping which associates with each variable v a value $v from Valv. In the (cartesian product) space spanned by sets Valv as 'coordinates', known as the state space, a state is a 'point' determined by selecting an element, $v, on each axis Valv. Conversly, given a state, one can think of values of all variables (in this state!) as the coordinates of the corresponding point.

Subsets of the state space are usually characterised by predicates, i.e. functions that map states to boolean values {true, false}. A state ($x, $y, ..., $z) satisfies a predicate P iff P($x, $y,.., $z) = **true**. Since the scope for confusion is minimal, the set of states which satsify a predicate will be denoted by the same symbol as the predicate.

When we consider a computation, a very important subset of the state space, characterised by a predicate OBSERVABLE, consists of all states than can be observed by an outside observer. We shall not analyse too carefully the exact meaning of the metastatement "a state can be observed by an outside observer"; suffice it to say that if the state ($x, $y, ..., $z) can be observed the values of its individual coordinates can be read out by the observer in question. For a state outside the OBSERVABLE set, values of individual coordinates may, but need not, be available. Unless explicitly mentioned, all named sets in the sequel of this paper are subsets of OBSERVABLE.

Let s be a state transformation, i.e. a map from states to states. Denote by **Doms** and **Rans** the domain and range of s, respectively. For ($x0,$y0,...,$z0) in **Doms**, s($x0,$y0,...,$z0) denotes the application of s to state ($x0,$y0,...,$z0). A state transformation with non-empty domain and range is called a *well defined state transformation* (wdst) iff its application to an observable state yields an observable state.

An obvious wdst is the identity state transformation, skip, defined by

```
skip(x,y,...,z) = (x,y,...,z)
```

Let p,q,...,r be wdst, and P,Q,...,R be predicates such that P implies Domp, Q implies Domq, ..., R implies Domr. We say that

```
(x:Typx,y:Typy,...,z:Typz)/($x,$y,...,$z)
            P -> [p]
              ||
            Q -> [q]
              ||
              ...                                    (1)
              ||
            R -> [r]
              ||
~(P or Q or ... or R) -> [skip]
```

specifies a single-processor computation with global variables x,y,...,z, initialised in state ($x,$y,...,$z).

Let progp, progq,..., progr be programs correctly implementing wdst p,q,...,r in the language of [DIJ]; in addition to the global variables x,y,...,z, each program may also employ local variables (whose sets are denoted by Locvarp, Locvarq, ..., Locvarr, respectively), distinct from global ones. It is assumed that the termination condition for progp is implied by P,

```
P implies wp("progp", TRUE)
```

(similarly for progq and Q, ..., progr and R).

The computation specified by (1) is then implemented by

```
x,y,...,z:=$x,$y,...,$z;
DO
            P -> progp
              []
            Q -> progq
              []
              ...                                    (2)
              []
            R -> progr
              []
  ~(P or Q or ... or R) -> skip
OD
```

The computation (2) is certainly non-terminating, it may also be non-deterministic. The possible nondterminism of (2), arising form some P,Q,...,R not being pairwise disjoint, is fully compatible with a fairly conventional view of computing. The built-in non-termination ('endless repetition of skip') represents a view of 'remaining in an observable state that does not satisfy any of P,Q,...,R.

In order to avoid some inessential but tiresome complications in the remainder of these notes, it is assumed that the global variables are passed to progp, progq,..., progr in the manner of Algol-60 procedural parameters called by value. Thus it is assumed that **Locvarp** includes variables locx, locy,..., locz, different from any local variables created for 'internal purposes', {u,v,...,w} are the only write-accessible global variables (their value may change as a result of progp exection), and progp is of the form

```
progp: locx, locy,..., locz:= x,y,...,z;

       other  statements of progp in which
       no assignements to global variables      (3)
       x,y,...,z are made (assignements to
       locx, locy,..., locz are allowed!);

       u,v,...,w:= locu, locv,..., locw
```

Thus it is only the last line multiple assignement of progp which changes the (global) observable state. Also for simplicity of further exposition it is assumed that both multiple assignements (first and last lines of (3)) are atomic and instantaneous; in making this assumption we explicitly ignore any implementation issues. The 'other statement' part of progp shall be referred to as *body* of progp. (Similar conventions apply to progq,..., progr.)

Returning now to the the specification (1), observe that it can be seen as an association of transformations with (sets of) states. This illustrates, in the familiar frame of reference of single-processor computation, the pragmatic essence of the proposed approach to specification. Rather than being goal-oriented, the proposed specifications are reaction-oriented. The notion of a 'state' was so far understood as a merely convenient abbreviation for 'an element of cartesian product space of value sets of variables'. If this notion is now interpreted as 'a state of a (real) system', we may say that in situations characterised by P,Q,...,R (and regardless of any 'history', cf. [JON]) the system 'reacts' by executing transformations p,q,...,r; in all other situations it remains in the same observable state. Of course, if the system has but one active element (processor) capable of actually performing the program that implements the transformation, cf. (2), the system's state changes only with the completion of this program execution, cf. (3). In fact, as far as changes of observed states are concerned, with a single processor there is absolutely no difference bewteen 'instantaneous' and 'protracted' execution of a transformation.

In a multiprocessor system a specification of the form (1) would be entirely inadequate. Indeed, if among the predicates P,Q,...,R there is at least one not disjoint pair, the system may find

itself in a state in which two guards of corresponding program (2) would be **true** and therefore two processors may start simultaneously from the same state. Thus a single observable system state could have two, possibly different, actual images under specified computation; two possibly different observable system states could arise from one. This is not a mere extension of the nondeterminacy of choice: we are dealing here with a possible *fission of the system*. Moreover, in the presence of two or more processors, this unpleasant complication cannot be removed by the simple expedient of making the guards mutually exclusive. Indeed, if the performance of the transformations is not instantaneous, two processors may start executing the same 'enabled' transformations (becuase the enabling state 'lasts') and, if they do not progress at the same rate (why should they?), the same image state will be established twice or more times, but at different instants, which again amounts to splitting the actual system into independently evolving 'copies'.

The traditional remedy consists in introducing some sort of 'traffic regulations' for processors (or, equivalently, some constraints on processes) which exclude, suspend or otherwise restrict mulitplicity of transformations if it only could lead to state-fission nondeterminism. In other words, the conventional remedy is *preventive*; as a consequence, a number of processors may have to be temporarily idled; with conventional approach, the price of elimination of uncontrolled indeterminism is an underutilisation of multiplicity of procesors.

It is tempting to consider an exactly opposite policy: allow full utilisation of *all* available processors (including infinitely many!) by permitting *all* transformations associated with *all* satisfied predicates to be performed simultanously and in as many copies as the available number of processors allows, but *restrict admissibility* of resulting state changes. This may be achieved by specifying the conditions under which the completed transformation is accepted; a transformation completed under conditions that fail to meet the specification will be voided. Figuratively speaking, the proposed remedy is *curative*, and thus strongly related to ideas of fault-tolerant computing, cf. [RAN].

In the presence of multiple processors (active agents) one is no longer assured that the system state will remain unchanged while a selected wdst is being performed. The implementation (3) of a transformation p guarantees that the body of local computation defined by progp, is insulated from any (adverse or otherwise) effects of state changes resulting from concurrent actions of other processors; thus if the predicate P properly describes the sufficient conditions for progp execution, it can be safely executed. But when the local computation is completed, its results may no longer be desirable (appropriate, acceptable, ...) in the system state that may have by then arisen. Therefore we introduce the second predicate, *postguard* (and rename the first one *preguard*) into an elementary block of specification. Thus

```
(P0,P1) -> [p]                (4)
```

specifies that in states that satisfy the preguard P0 the system reacts by transformation p which, however, is effective only in states satisfying the postguard P1. If, when the transformation p is completed, the system does not satisfy P1, the transformation is voided. Naturally, when there is only one processor, nothing can change the system state during the execution of p and (4) collapses to

```
P0 and P1 -> [p]          (5)
```

The comparison of (5) and (4) suggests that the postguard describes such aspects of the conditions under which p is considered appropriate that are not essential for the local computation implementing p, but relate to its 'global' acceptance. One can say that the guard is split in two parts: the preguard which enables p, and the postguard which makes p acceptable. The evaluation of postguard *may* be postponed if it is expected that (due to the actions of other agents) the acceptability of p will have been established by the time p will be completed. The evaluation of postguard *should* be postponed if there are reasons to suspect that the actions of other agents may render p unacceptable while it is being executed. Naturally, in the case when the transformation p can (should) be performed regardless of any other action, one can use the always satisfiable postguard, TRUE. Thus

```
(P,TRUE) -> [p]           (6)
```

specifies such a transformation (its 'collapsed' form, P and TRUE -> [p], contributes nothing to our discussion).

It is easy to modify the implementation (3) to cater for the postguard:

```
progp: locx,locy,...,locz:= x,y,...,z;
       body of progp;
       IF P1(x,y,...,z)  ->  u,v,...,w:= locu, locv,..., locw
                         []
          ~P1(x,y,...,z) -> skip
       FI
```

or, even more succinctly,

```
progp: locx,locy,...,locz:= x,y,...,z;
       body of progp;                                    (7)
       if P1(x,y,...,z)
         then u,v,...,w:= locu, locv,..., locw
```

As before, the multiple assignements in the top and bottom lines are assumed to be instataneous; the predicate P1 is evaluated in the current state, i.e. the values of variables x,y,..,z may differ from those that have been used in the top line multiple assignement.

With similar modifications to progq,...,progr, the specification

```
(x:Typx,y:Typy,...,z:Typz)/($x,$y,...,$z)
          (P0,P1) -> [p]
                  ||
          (Q0,Q1) -> [q]
                  ||
                  ...                                    (8)
                  ||
          (R0,R1) -> [r]
                  ||
(~P0 and ~Q0 and ...~R0,TRUE) -> [skip]
```

defines a multiprocessor computation eventually implemented by local computations progp, progrq,...,progr. Whenever a preguard is satisfied and there is an available processor, the corresponding local computation may be started and no assumptions are made on the choice of local computations to start when several preguards are satisfied in a single state. Similarly, no assumptions are made with respect to the speed with which individual processors execute local computations, but it is assumed that each processor is capable of executing any local computation. If only one processor is available, the specification (8) is entirely equivalent to a specification of a single-processor computation with ('collapsed') guards defined by conjunction of respective pre- and postguards.

Experience with using the proposed style of specification indicates that a quite often needed kind of postguarded transformation is of the form

```
(P,P) -> [p]
```

indicating that the transformation p is acceptable if the system state, insofar as captured by the guards, does not change during the (body of a) local computation that implements p. An obvious implementation policy would be to establish a one-bit trap on assignements to variables on which P is defined, thus saving the evaluation of P if no assignements occur while the body of progp is executed.

In these notes we ignore all such considerations on practical implementation.

It is perhaps interesting to observe that the inclusion of postguards in the specification correlates with the preferred (by some physicists) style of description of experiments in quantum physics. For instance, in the delayed-choice split-beam experiments (cf. [WHE]), photons (or electrons) are made to travel through an experimental apparatus along *one* of two routes, routeA or routeB, or along *both* routes simultaneously. The discrimination between the two options is effected by the sensing device placed at the endpoint of both routes. The sensing device may be freely chosen from two available: DEV1 and DEV2. When DEV1 is used, the photons appear coming only by one route, when DEV2 is used they appear coming by both routes. The apparent ability to

change the history by selecting the sensing device *after* a photon has been fired and thus started its travel either by one route, or by both routes simultaneously is a little unnerving to some physicists (and many philosophers). We could specify such an experiment in a very simple way:

```
(FIREABLE, DEVIP1) -> [go by one route]
                   ||
(FIREABLE, DEVIP2) -> [go by both routes]
                   ||
(¬FIREABLE, TRUE)-> [skip]
```

where FIREABLE describes states in which a photon can be fired, and DEVIPi is satisfied iff device DEVi is in place. Note that with a single processor and collapsed form of specification we need to know which sensing device is in place ***before*** the photon is sent on its travel; in a multiprocessor environment we get an exact replica of the delayed choice experiment!

Let us now consider a few examples of the proposed style of specifying multiprocessor computations.

EXAMPLE1.

(In this and subsequent examples the transformations are described by pseudoprograms; only the essential global variables are listed; predicates often are given 'telling' names without providing relevant formulae.)

Consider a bank with multiple tellers accepting payments and paying out cheques. The specialty of the bank is that it pays out only in dimes; thus when presented with a cheque for, say, one hundred dollars, the teller must count one thousand coins, a rather long process. To pay out, a teller must be satsified that a cheque has been presented and that the corresponding account is sufficiently in credit to honour this cheque. Upon presentation of a cheque, the teller may check the balance of the account and finding it sufficiently large, start counting the coins. Lest, however, actions of other tellers reduce the balance while he assembles the requested number of dimes, the teller must suspend these actions (or, avoiding a total blockade, must suspend the paying-out actions on a given account). If the actions of an individual teller are not to interfere with actions of all tellers, the balance must be checked at the same instant as the bag of coins is handed out. The specification

```
(request>0, balance>request and payout=0) ->
                        [count_coins;
                         payout,request:=request,0]
                   ||
(payout>0, TRUE) ->
                        [balance:=balance-payout;payout:=0]
```

describes a non-interfering paying-out procedure (request, payout and balance are variables relating to the same account; for simplicity, the account-identification has been left out).

Note that the protracted action (coin counting) is not delayed awaiting the balance to be sufficiently large; it is undertaken as soon as a request is presented. If, by the time the coin counting is finished, the balance is insufficient or a previous payment from the same account has not been completed, the count is 'voided' and a new counting to meet the same request will start. Actually, many countings for a single request may be in progress simultaneously but only one can succede in establishing positive payout (thanks to the second conjunct in the postguard). True to our philosophy, no explicit sequencing (nor timing!) is present in the specification. The price for this simplification of the design is the potentially wasted work of a processor (or processors) counting coins in vain. This is how we 'trade clocks for chips'.

EXAMPLE 2.

Assume that in the system under specification, some phenomenon manifests itself by a change of value of a global variable x. Only two values of x are admissible, x = 0 and x = 1. Following is a skeletal specification of the system, where the only explicitly listed parts specify a counter of the occurrences of the phenomenon; as far as the counter is concerned, the occurrences of the phenomenon are entirely spontaneous:

```
{x:0-1,n:integer,...}/{0,0,...}
(x=0, x=1) -> [n:=n+1]
           ||
(x=1, x=0) -> [n:=n+1]
           ||
          ...
```

Note that in this case the collapsed guards are identically **false**, which nicely corresponds with the fact that the problem is totally meaningless in a single-processor environment. (If a specification includes (P,Q) -> [s] with contradictory P and Q, this specificaton cannot be implementen in a single processor environment.)

In some problems, where a single-processor implementation does not make sense, it may be useful to mark parts of the specification as meant for a dedicated processor. Such is the case of specification of Example 2. Using the pair of brackets ⦃ and ⦄ for marking, the specification may be rewritten as

```
   {x:0-1,n:integer,...}/{0,0,...}
⦃ (x=0, x=1) -> [n:=n+1]
           ||
(x=1, x=0) -> [n:=n+1] ⦄
           ||
          ...
```

EXAMPLE 3. (Dining philosophers)

The notation in this example is localised to a chosen philosopher ('as seen by him'). The meaning of predicates is as follows

H - the philosopher is hungry
LOT - the left fork is on the table (free for taking)
ROT - the right fork is on the table
PL - the philosopher possesses the left fork
PR - the philosopher possesses the rigth fork

The transformations are specified by means of the following actions, listed alongside the effects their completion has on predicates:

take_l	makes PL **true** and LOT **false**
take_r	makes PR **true** and ROT **false**
release_l	makes LOT **true** and PL **false**
release_r	makes ROT **true** and PR **false**
think	has no effect on values of 'fork' predicates
eat	has no effect on values of 'fork' predicates

It is assumed that each philosopher is to be implemented by a dedicated processor

```
{ (~H,TRUE) -> [think; make_H_true]
                ||
  (H and LOT,LOT and ROT) -> [take_l]
                ||
  (H and ROT,ROT and LOT) -> [take_r]
                ||
  (PL and ROT,ROT) -> [take_r]
                ||
  (PR and LOT,LOT) -> [take_l]
                ||
  (PL and PR,TRUE) -> [eat; make_H_false; release_l; release_r]
                ||
  (PL and ~(ROT or PR),TRUE) -> [release_l]
                ||
  (PR and ~(LOT or PL),TRUE) -> [release_r]
                ||
  (H and ~(LOT or ROT)) -> [skip] }
```

Note that the preguards in the philosopher-process are not mutually exclusive; it is assumed that the choice (and with a single processor "serving" this process a choice must be made!) is totally nondeterministic.

Contrary to Example 2, it would be possible to implement the dining philosophers by a single processor, as evidenced by the not necessary falsity of collapsed guards. Indeed, a simple analysis shows that the specification resulting from collapsing the guards and introducing a uniform notation (unscrambling the

aliases) can be implemented without deadlock by a random selection of transformations guarded by satisfied predicates. Thus the indication of dedicated processors in this example is not a necessity but rather a mere emphasis of the original problem statement. (In a full-scale specification language, it would be probably adviseable to differentiate between the necessary and merely convenient dedication of processors.)

EXAMPLE 4.

Consider a process that reads a (presumably infinite) file, record by record, and processes each record as soon as it is available. Actions needed to read and process a record are specified, respectively, by read_next_record and process_record. States in which a record is available for processing satisfy the predicate WORK. Due to actions of other, concurrently executing processes, the observable state may change at any instant to one satisfying the predicate INTERRUPT. The considered process must then first react_to_interrupt and then return to its "normal" cycle without disturbing the sequentiality of its main task. Such a process may be specified as follows

```
{ (~WORK and ~INTERRUPT,~INTERRUPT) ->
                              [read_next_record; make_WORK_true]
            ||
  (WORK and ~INTERRUPT,~INTERRUPT) ->
                              [process_record; make_WORK_false]
            ||
  (INTERRUPT,TRUE) ->
            [react_to_interrupt; make_INTERRUPT_false] }
```

Note that although this process may be executed by a single, dedicated processor (as indicated), unless the system has other active agents no interrupts can occur when the initialisation of variables is such that WORK and INTERRUPT are false. Note also that the specification requests that an interrupted action (reading or processing) is voided, thus as soon as the interrupt is handled (an unconditional action) the same action that was interrupted (and voided) is resumed, thus assuring the sequentiality. The above specification makes no allowance for hierarchy of interrupts, nor for multiple interrupts occuring in quick succession. Both refinements can be easily dealt with in the proposed style, but would require specific assumptions about the hierarchy of interrupts.

EXAMPLE 5. (Producer/Consumer exclusion)

This example is presented as a pair of processes (with two dedicated processors). In a standard set-up it is assumed that portions are produced by one process and consumed by the other. Portions are passed from the producer to consumer via a shared buffer, capable of holding N+1 portions. WORK is a predicate that charcterises the states in which consumer process can perform some useful work on a portion; this work is represented by the

action consume_portion. Predicate READY_TO_PUT characterises states in which producer process is ready to pass a portion on the buffer; producer's action produce_portion produces a portion. Two actions, put_portion and take_portion, specify the actual transfer of portions to and from buffer, respectively. Note that putting and taking are two boolean-valued variables. The system is supposed to be initialised in a state in which both WORK and READY_TO_PUT are **false**.

```
{k:integer, putting,taking:boolean,...}/{0,false,false,...}

{ (~WORK, k>0 and ~putting) -> [taking:= true]
                            ||
  (taking, TRUE) ->
       [take_portion; k,taking:= k-1,false; make_WORK_true]
                            ||
  (WORK, TRUE)    -> [consume_portion; make_WORK_false]
                            ||
  (~WORK and k=0, TRUE) -> [skip] }


{ (~READY_TO_PUT, TRUE) ->
       [produce_portion; make_READY_TO_PUT_true]
                            ||
  (READY_TO_PUT, k<N and ~taking) -> [putting:= true]
                            ||
  (putting, TRUE) ->
       [put_portion; k,putting:= k+1,false;
                     make_READY_TO_PUT_false]
                            ||
  (READY_TO_PUT and k=N, TRUE) -> [skip] }
```

REFERENCES

[DIJ] E.W.Dijkstra: *A Discipline of Programming*. Prentice Hall, 1976.

[JON] C.B.Jones: *Systematic Software Development Using VDM*, Prentice Hall, 1986.

[RAN] B.Randell: System Structure for Software Fault-Tolerance. *IEEE Trans. on Soft. Eng.* **SE1** (1975), 220 - 232.

[TUR] W.M.Turski: Time Considered Irrelevant for Real-time Systems. *BIT* **28**(1988) 473 - 486.

[WHE] J.A.Wheeler: The Computer and the Universe. *Int. J. of Theor. Physics* **21** (1982), 557 - 572.

DISCUSSION

First Lecture

Rapporteur: Marta Pietkiewicz-Koutny

Professor Kopetz suggested that the dining philosophers problem could be solved by introducing timeouts. Professor Turski answered that that would imply reading of a global variable making the solution unacceptable.

It was then suggested that time might be introduced to the solution by a special continuously counting process. The speaker replied that he was able to talk about the problem and its solution without any reference to time, and there is no place for time in his framework. Professor Turski also stressed that he was only talking in terms of changing the state of the system in some specific states.

Professor Nehmer said that in the formal framework described during the lecture one cannot express changes over a long period of time, e.g. a week. Professor Turski emphasized that in the original statement of the problem one is not concerned with such issues, only with eating, being hungry, starving, etc. A question was asked what happens if the counter is not able to recognise the changes of variables due to, e.g., their higher cycle rate. Professor Turski answered that such changes are simply ignored; something which cannot be observed 'does not exist'.

To a question whether properties such as liveness and deadlock-freeness can be discussed within the presented framework, Professor Turski replied that these are properties of traces, and cannot be discussed within his model.

Second Lecture

Rapporteur: Maciej Koutny

The speaker was asked whether he would only measure variable quantities different from time. Professor Turski answered that in the kind of problems he has been discussing, time is not inherent to the problem, it has rather been introduced to avoid addressing more fundamental questions related to the problem.

Professors Anderson and Carter asked why redundancy techniques cannot be used for the verification of astronomical time within a computer system. The speaker explained that no program is in principle able to verify internally the absolute time, and contrasted this with an apparent ability to verify the consistency of bit patterns stored in the computer's memory. To a suggestion that time could be measured by counting the clock's signals, Professor Turski replied that in his view this would not solve the problem either, as we can never be certain that the rate at which the clocks work are indeed correct.

A question was then asked whether one can justify the use of timeouts in a stochastic environment. The speaker answered that such an approach cannot be accepted as valid for highly critical systems. He also stressed that a right approach would be either to wait infinitely long, or to carry out further processing and employ interrupts.