CONTROLLING DISCRETE REAL-TIME SYSTEMS WITH PARALLEL PROCESSES

C BRON

Rapporteur: A Tully

CONTROLLING REAL-TIME SYSTEMS WITH PARALLEL PROCESSES

Coen Bron
Department of Computing Science
University of Groningen
PO Box 800
9700 AV Groningen
Netherlands

Abstract
In this paper it is argued that the automatic control of industrial systems can be based on a structure of co-operating sequential processes. This form of control is obtained by means of a tight coupling between the (computer based) controlling processes (tasks), and the (physical) processes being controlled. The real-time aspects of this control merge naturally with the overall structure. A software package (ROSKIT) is described which supports this approach.

1 Introduction

This lecture is mainly concerned with the structure of computer control of mechanical processes which are a part of industrial systems. The epitheton 'real-time' stems from the requirement that there is an upper limit on the amount of (real) time that may elapse between the request for a controlling action and the moment the controlling action actually takes place. It is generally attempted to construct systems in such a way that if dead-lines can not be met, an error state will result, rather than the occurrence of a catastrophe. However the way in which the system may recover from an error state (e.g. by human intervention) may be such that failure to meet real-time requirements causes a serious, *abrupt* degradation of the performance of the controlled system or a subsystem within. This contrasts with (e.g.) a time sharing system where the real-time aspects can not be neglected altogether, but the degradation of the system's performance as witnessed by the user (as in the case of the echoing of characters by a screen oriented editor) is gradual.

In defining the setting of this lecture we have deliberately used the term *mechanical processes* to suggest the time scale of responses to be of *ms*-scale rather than of μs-scale. These figures should not be taken as absolute, but they indicate that the time-grains of controlling system and the system to be controlled must lie a few orders of magnitude apart. This requirement is obviously satisfied by the time-grains of many industrial processes and common present day computing equipment. Conversely: the speed of present day computing equipment is probably insufficient for the described method of control to work in (e.g.) high speed physics experiments, or real-time image processing. In the first case the raw computing speed may be the limiting factor. In the second case it will be the amount of computing involved in image processing.

On the other hand the term *mechanical processes* must be understood in a sufficiently broad sense and the sequel will —hopefully— make clear that the control of continuous (chemical) processes may also be covered by the contents of this lecture, provided these processes can be discretised on an acceptable time scale. The term *industrial* must be understood similarly, to include systems for handling, moving, storing and fetching objects or raw material.

2 Background

A traditional way of controlling (mechanical) systems is the use of PLCs (Programmable Logic Controllers). This form of control is based on relays and (binary) inputs. The control can be described in terms of logical equations that must be satisfied by the controlled system. A common form in which these equations are represented is by so-called ladder diagrams, that show an analogy with electrical circuit diagrams, an AND being represented by two on/off switches connected serially, an OR by two switches in parallel.
E.g. the equation for output A, might read

$$A = I2 \vee (A \wedge \neg I1)$$

which is represented in a ladder diagram by Figure 1:

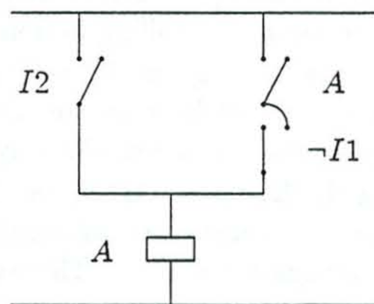

Figure 1: State depicted: A=false, $I2$=false, $I1$=true

In words: output A should be activated either if it is already *on* and $I1$ is *off*, or whenever $I2$ is *on*.

Based on a set of logical equations a corresponding program for the actual controller must be derived. These programs are (again traditionally) written in a symbolic machine code, viz. as a control loop, which —in forward scan through the text— must assure that all outputs satisfy the given equations.

The structure of the software for such a system takes the form of a cyclic executive [2,8]. Whereas such an executive may have its merits if the schedule can be determined statically (as may be the case when processes initiate periodic actions only), it seems quite unfit when the sequence of control events is fully driven by the

progress of external processes. In these cases the corresponding PLC programs are quite contorted.

The computing scientist looks at this form of controlling ever more complex systems with horror. It lacks the basic structuring facilities (like subroutines) that we have come to regard as essential in managing complex tasks and the tools to generate the control programs from a high-level description (editors, compilers,...). Only very recently proposals are being put forward to fill in these deficiencies[5,7].

Another, even more serious drawback is the lack of scaling. I.e. in designing ever more complex industrial systems, the set of equations describing the behaviour of these systems may well fall victim to a combinatorial explosion. At the same time the testing of systems controlled by PLCs will be virtually impossible because of the impossibility of driving the system into all possible states.

3 Decomposing the systems to be controlled

Whereas in PLC programming the system to be controlled is described by a set of logical equations that must be simultaneously satisfied, we may also view that system as the set of ongoing activities being executed as part of the accomplishment of the system's task as a whole.

This situation is quite comparable to the one we have within an operating system which has the task of controlling several simultaneous operations. In this area it has been established long ago that a valid structuring concept is the decomposition of such a system in a number of co-operating sequential processes [4]: each process describing a time ordered sequence of activities, where due to the particular physical constraints (resources, producer-consumer relationships) progress of these processes may have to be delayed by means of synchronizing interactions.

This analogy suggests that the control of the industrial systems we are concerned with might be structured along the same lines, but for the real-time aspects of this control. On closer look these aspects are not of such a nature that it would upset the above decomposition. It only needs to be guaranteed that the controlling system reacts timely to the requirements of the system to be controlled, and furthermore, where timing comes into the picture, one may view the progress of time as a separate process that —by its synchronizing actions— may control the progress of other time-dependent processes.

The decomposition of an industrial control system into parallel processes is by no means unique, because the time-ordering constraints on the individual actions induce a partial ordering only, so there will be many ways in which the system can be decomposed into a set of total orderings plus synchronizing operations.

For a (manufacturing) plant there are two rather obvious ways to perform this decomposition viz. either by relating sequential processes to (parts of) the machinery that —by its physical properties— must perform some actions in sequence, or by relating sequential processes to items (products) that pass through the physical system.

Although an object oriented approach might suggest the second decomposition,

the first one has the advantage of being more static: the processes controlling the system are created when the system is instantiated (started) and all perform an endless duty-cycle. Furthermore, the object oriented style might break down (or at least loose some of its attractiveness) if physical objects are assembled from parts.

It might be argued that in stead of decomposing a physical system to be controlled in a number of sequential processes, mutually synchronized where necessary, one should design the complete system (both the physical one and its control) with this decomposition principle as a guideline. This is certainly true, and the overall design will benefit, but quite often one has to implant automatic control on a physical system that is already (partially) in operation, or the components of which have already been designed and/or constructed. Experience has shown that even then, a collection of sequential processes may be the basic form of control. It need hardly be argued (at this seminar) that designing the control system as a collection of sequential processes allows us to reason about these processes with the same mental tools as have proved vital in reasoning about Operating Systems and other parallel systems (e.g. Gries-Owicki) and therefore no new formalisms or patterns of reasoning are required.

So far, so good, but where does the real-time aspect come in? Well, it must be guaranteed that at any instant in time (take or leave an acceptable margin depending on the physical properties of the system being controlled) the state of the controlling process is equivalent to the state of the process being controlled.

And this —of course— can only be accomplished if the internal and external processes can exchange signals that synchronize the two at those instants in time where this is relevant. As an example we take the control of a lever moving from A to B assuming the lever to be initially in position A and the controlling proces in a corresponding state:

```
move-from-A-to-B:
 {lever in position A}
 start_lever_motion     ;  {generates an output signal}
 wait_for_position_B    ;  {observe an input signal}
 stop_lever_motion      ;  {generates an output signal}
 {lever in position B}
```

The internal state "waiting_for_B_detection" corresponds to the external state of the lever moving somewhere between A and B, and so we see a whole collection of external states being mapped onto one internal state since the details of the external state (the exact position of the lever) are irrelevant to the controlling proces. For this form of control to show the desired real-time behaviour, it is obviously necessary that "wait_for...." is not implemented as a form of busy waiting.

Now if we assume the controlling processor(s) to have sufficient capacity to schedule all active control processes within the period of time considered an acceptable margin it will be clear that the controlling processes are always in phase with the processes being controlled. There will be no need to determine a static schedule for a cyclic executive since the processor(s) can cope with the dynamic behaviour of the system.

Up to this point we were only concerned with the matching in time of internal and external process. Now we add the second aspect of real-time systems (which as a

matter of fact is found in Operating Systems as well), concerning controlling actions that have to take place at a specified time.
Examples of such actions are:

- the periodic reading of sensors and possibly performing of a controlling action
- an action to be performed after a certain amount of time has elapsed (e.g. the opening of a valve some time after heating has stopped). Such a timed action may sometimes replace an action triggered by an external signal (such as the temperature having reached a certain level).
- a control action that has to take place at a specific instant (e.g. it may have been specified by an operator).

All of these actions imply that the controlling process has to wait until the lapse of a certain amount of real time, and so we can synchronize the process with a time process which acts as the provider of an external signal as soon as the waiting period has expired.

A third aspect of real-time, the detection that external processes do not show the required behaviour when they fail to provide a signal within a predefined amount of time, the so called 'time-out', will be dealt with in section 5 on error handling.

In summary: The implementation of the computer control of a physical system may be done by first decomposing the system into a set of sequential processes. These processes will very often found to be cyclical. Next we allocate to each physical process a control process that runs in parallel and fully synchronous with the physical process. This synchronization is accomplished by the exchange of signals between the controlling process and the controlled process. Viewed from the controlling process, the input signals serve to synchronize the controlling process with the physical process. The output signals implement the actual control. Any synchronization that is required within the physical system can thus be delegated to the internal synchronization of the controlling processes, giving due care to 'stop' the physical process if the controlling process is about to be blocked. Apart from the strict synchronization between controlling and controlled process any further real-time constraints can be met by implementing time as a process that performs the necessary synchronizing actions (waking-up or timing-out controlling processes) at the required instant. The role of the time process is very analogous to that of the scheduler in an event driver simulation.

4 A software package for real-time control

In this chapter we give a brief outline of a set of software modules with which the above form of control can be (and indeed *is*) accomplished. It should serve as an example or as a demonstration of the viability of the approach described rather than as the ultimate word in real-time systems.

ROSKIT (Real-time Operating Systems KIT) is a set of modules supporting the implementation of industrial systems for real time control. It has been based on

MOSKIT (Modular Operating Systems KIT), a set of modules from which portable Operating Systems can be constructed [6]. The main contribution of ROSKIT is the introduction of time as an essential factor of control, and a very basic mechanism for receiving and producing external signals. ROSKIT is not an Operating System providing an environment within which applications can be run, and —additionally— real-time oriented system calls.

It is just a set of modules that are integrated with the application and finally down-loaded into a control processor. Within that application they *do* provide the functionality typically provided by an operating system kernel, like interrupt handling, parallel processes and their scheduling, synchronization primitives, basic i/o-functions. For this arrangement we like to use the term *Tailor-made Operating System.*

ROSKIT has been written in Modular Pascal (ModPas for short)[9] a version of Pascal providing separate compilation of modules and a tighter control over visibility and accessibility than the classical block structure of monolithic Pascal. Therefore, a ROSKIT control application may be viewed as a single, *modular* program with the full advantage of block structure and locality of process declarations, including the access restrictions imposed by the modular structure [3].

Interprocess communication may therefore be based on a shared memory model, which proceeds at speeds as are typically required in a real-time environment. Where necessary, interlocks must be explicitly programmed, although ROSKIT does provide the functions that make this an easy task.

Since ModPas is a strictly sequential language, based (as Pascal) on a single stack model, ROSKIT must provide functions to define processes, i.e. set up a stack for a (new) process, and for context-switching (stack switching). In order not to burden the application-programmer with the necessary house-keeping, the ROSKIT development environment provides a preprocessor that accepts ROSKIT source modules (that contain declarations of parametrized process templates) and transforms them into the corresponding piece of ModPas text.

The language extension thus obtained consists of a process-declaration, which is syntactically equivalent with a procedure declaration, but for the reserved words PROCESS, BEGINPROC and ENDPROC replacing PROCEDURE, BEGIN of body, and END of body respectively.

The preprocessor transforms this declaration into that of a function which —when called— spawns a separate process executing the process body and delivers an identification of the newly spawned process (of `TYPE proc`).
Example:

```
    PROCESS controller (..formal params..)
    BEGINPROC..........................ENDPROC;
    ............;

BEGIN start ( controller(..actual params..))
;     start ( controller(..other(?) actual params..))
;     ...........
END.
```

In the above example both processes proceed anonymously, they may however also be instantiated as:

```
VAR c1, c2: proc;

BEGIN c1 := controller(..actual params..)
;     c2 := controller(..other(?) actual params..)
;     ...........
END.
```

As matter of fact, all `start` does is the voiding of the function-result yielded by the call of `controller`. We will now look at the ROSKIT modules in somewhat more detail.

4.1 An overview of ROSKIT modules

BASYS (BAsic SYStem)
This module, or a variant of it, is present in any ModPas environment. It provides for the dynamic linkage of modules and the setting up of the addressing environment for each module (see [3]). It might be considered the bootstrap module for any ModPas program.

PARLEL
The functions of this module can be grouped as follows:

- creation (*fork*) and termination of processes. (Note: just before termination a process must perform a *join* with all the processes it instantiated (children), since they may share the adressing environment with this process (father).) The correct behaviour of a process is enforced in PARLEL as a consequence of the way in which processes are spawned.
- synchronization of processes.
  Several levels can be recognized
  - mutual exclusion during process switching
  - manipulation of process-queues (only used internally)
  - semaphores: `init`, `P`, `V` and

    ```
    FUNCTION P_if_free(VAR s: semaphore): boolean;
    ```

    Semaphores are by far the most used mechanism for synchronization of controlling processes. A possible explanation of their popularity might be that in the systems being controlled a good deal of the parallellism is obtained by buffering mechanisms (like conveyor belts, pallets etc.) and semaphores are pre-eminently suitable to synchronize producer/consumer relationships.

    The additional primitive `P_if_free` derives its existence from the requirement of strict synchronization between internal and external process.

Since the performance of a P-operation represents potential blocking, it must be possible to block the external process only if the P would result in blocking the internal process. This gives rise to the following program scheme:

```
IF NOT P_if_free(s) THEN BEGIN
   deactivate_external_process
;  P(s)
;  reactivate_external_process
END
```

– On the last level we have conditional critical regions, consisting of a

```
TYPE ccr = ?;
PROCEDURE ini_ccr(VAR c: ccr);
PROCEDURE enter_ccr(VAR c: ccr);
PROCEDURE leave_ccr(VAR c: ccr);
PROCEDURE await_ccr(VAR c: ccr; FUNCTION cond: boolean);
```

- PARLEL also defines a way to connect a parameterless procedure to an interrupt source, thus installing a programmer defined interrupt-handler.
- raising of an exception by one process in another (see section 4).

BITIO

This module provides the basic communication with external processes in the form of bit-values that can be set, read and waited for. These signal bits are numbered from zero upwards and mapped onto hardware locations in a machine dependent way. Relevant bitnumbers will ordinarily be defined as named constants. This module provides:

```
TYPE io_bit = 0 .. ??;
FUNCTION get_bit(i: io_bit): boolean;
PROCEDURE put_bit(i: io_bit; val: boolean);
PROCEDURE wait_for(i: io_bit; val: boolean);
```

CLOCK

This module defines time as a process which is periodically reactivated by the clock interrupt, and which runs at the highest priority. The frequency of the clock interrupt determines the basic real-time response of controlling processes, but —where necessary— a faster response may be obtained by installing specific interrupt handlers to service those parts of the controlled system that require a faster response. (Such interrupt handlers typically provide a V-operation on a semaphore, therefore they do not conflict with the overall structure of the controlling system.)

The tasks of the time process are the following:

- updating of the global system clock (keeping date and time)
- waking up processes whose requested delay has expired

- waking up processes in cases that an input bit has acquired the value being waited for (see BITIO). This obligation actually involves a scanning loop, but its overhead is reduced because

    1. only those inputs are scanned for which there is an outstanding wait
    2. all scanning is done in one process instead of periodically waking up the waiting processes. Thus overhead in process switching is reduced.

- signalling the expiration of timers that have been turned on by processes and have not been turned off before their expiration. (See the section 5 on error handling)

More specifically this module provides:

```
- TYPE day_time = ??;
  PROCEDURE set_time(...); {for initialization}
  FUNCTION get_time: day_time;
  PROCEDURE time_string(CONST t: day_time; VAR tstring : STRING );
- PROCEDURE wait(nr_of_ticks: integer);
            {a programmed real-time delay}
- TYPE timer = ??;
  PROCEDURE new_timer(VAR t: timer; nr_of_ticks: integer)
  PROCEDURE stop_timer(VAR t: timer);
```

The last module in the ROSKIT package is a module LINEIO to share an operator console on a line-by-line basis, but since such communication is only done by processes with a low priority level, it does not contribute to the essence of real-time-control.

4.2 Scheduling

The scheduling algorithm applied within ROSKIT is both simple and effective:

- of all executable processes, one of the highest priority class is selected for the CPU and basically runs until it becomes blocked.
- If a process with a higher priority than the current one is unblocked it is assigned the CPU, otherwise the first rule would be violated.
- If a process relinquishes the CPU because of the previous rule, or if a process is unblocked that does not have a higher priority than the one currently selected for the CPU, it is entered at the tail of its own priority queue.

From these three rules we can derive fairness for the controlling processes due to the periodic activation of the time process (having a higher priority than the control processes). Synchronization between processes of equal priority does not give rise to unnecessary process switches.

For this scheduling to show the desired real-time behaviour it is vital that process switching takes place with a minimum of overhead. The general philosophy to

accomplish this is to have processes access data rather with a form of indirection than to make access as fast as possible at the expense of having to save and restore many process dependent data on process-switching. In the PDP 11-implementation, that context-switch consists of the saving (c.q. restoring) of two (stack-pointer) registers in (c.q. from) the process record.

5 Error handling

Compared to the situation in regular (everyday) programs, the situation in real-time control programs is in some ways simpler, and in some more complex.

On the one hand, control programs may be assumed to be both correct and robust and normally need not to bother about 'weird' inputs. I.e. in writing a control program there is a tight coupling between the design of the program and the way in which it is going to be used.

On the other hand, we must design control programs with an open eye towards any form of malfunctioning of the external processes and see to it that the control programs can cope with these external errors as integral part of the specification of their behaviour.

The handling of errors in a ROSKIT control program is based on the exception handling mechanism of ModPas, originally designed for the execution of a *single* sequential program. We briefly review:

- Exceptions are names of 'illegal preconditions' e.g. `x_index` is the name of the precondition that prior to an (array)-indexing operation the index value to be applied lies outside the permissible index range.

- Exceptions may be signalled upon detection of such illegal preconditions. (And for programs to be robust, the test on the legality of preconditions is a pre-requisite.)

- When an exception is signalled, all current blocks in execution (this is a dynamic nesting of blocks) are terminated upto and including a block containing a handler for *this* exception. The handler for an exception consists of a sequence of statements local to this block that is executed prior to termination of this block. A common form of exception handler is responsible for either the performance of some cleanup actions (finalization) or the generation of an appropriate error message.

- After an exception has thus been handled, the corresponding block is considered normally terminated and no trace of the exception is left unless it has been explicitly programmed in the handler. However, at the end of a handler the current exception may be reraised, (or an other one signalled) thereby propagating block termination further outward. Exception handlers with a reraise are typically connected to blocks with finalization obligations.

- A handler may either deal with a specific (named) exception, or with *any* exception. The construction

```
BUTFOR x_any: ..clean_up.. ; x-reraise DO
BEGIN ..block.. END
```

is typically used to handle finalization obligations.

In a control program consisting of several parallel processes, each process handles exceptions that are raised within these processes as described above, and it is exactly the general exception handling facility and the way in which it is used when new processes are created that prevents processes from going astray or terminating without fulfilling their termination obligations.

A new element is introduced by the fact that processes should be able to raise exceptions in other processes. For instance, in some kind of external emergency situation an exception must be broadcast to all processes concerned, upon which they should perform their corresponding handlers in stead of continuing their normal course. More in particular, the time process may have to raise a time-out exception in processes that have started a timer, and failed to stop it before it expired.

So we arrive at the following program scheme for endless control processes:

```
PROCESS controller (......);

  PROCEDURE do_control;
  ...local declarations...;
  BUTFOR x_any: cleanup; wait_for_reset; reinit_control DO
  BEGIN LOOP
          actual_control
        END
  END;

BEGINPROC init
;         LOOP
            do_control
          END
ENDPROC;
```

(In case init and reinit are the same, it can be moved inside the control loop of the controller process and removed from the exception handler.)

An analysis of the nature of external exceptions taught us that there is little sense in being too detailed in the (external) exceptions that can be raised by one process in another. One reason is that processes may share very little state and so there is no sense in detail. Another is the implicit asynchronism that is present in external exceptions. Whereas internal exceptions and the situations in which they may be raised can be traced by a judicious inspection of the text of a sequential program, external exceptions are fully impredictable and therefore can only be dealt with in a coarse way.

This analysis led to only two exceptions that make sense: One to abort the current operation of a process. What constitutes an operation is defined by the declaration

of handler for this abort exception. The other kills the process in which it is raised, i.e. it aborts all operations regardless of the presence of handlers. For this to be accomplished we had to define that the exception `x_kill` is always reraised at the end of a handler and cannot be overruled by the raising of another exception! In a ROSKIT application, where as a rule the process structure is static, and all processes are endless loops, the `x_kill` exception does not play an important role.

In ROSKIT the `x_timeout` exception is the third external exception, which will be signalled asynchronously if a timer expires before it has been switched off (stop_timer). On second thought, the connection of a timing facility with a potential wait (P or wait_for) to form a boolean function (cf. P_if_free) seems to do better justice to the principle of strict synchronization mentioned at the end of section 3.

The occurrence of external exceptions necessitates the concept of actions that must be indivisible with regard to the raising of external exceptions, usually to guarantee consistencies of coherent data (even if the data are fully local).

In ROSKIT such problems are solved ad hoc, in MOSKIT this has given rise to the concept of *immortality*. A section of code is immortal if it can not be interrupted by an external exception. Immortal sections can be nested.

Code executed as part of an exception handler is by definition executed as an immortal section. When an external exception is raised in a process during immortality, the exception is made pending until the process returns to a mortal state. It is superfluous to state that sparse use of immortal sections should be made in control applications that need a speedy reaction to emergencies.

The need for raising external exceptions is not always *directly* related to external circumstances. Indirectly, for example, a process (say A) that is involved in a communication with an other process (say B), may have to raise an exception in B to tell B that it cannot fulfill its communication obligations if the progress of A has been disturbed by external circumstances. In other words: some external circumstances may necessitate the broadcasting of an exception explicitly to all processes involved. (This *can* be done explicitly since processes can be started as named entities (see the example in section 4)).

Since, as we will explain in the next chapter, processes are the basic structuring mechanism, they may be internally decomposed into (sub)-processes. Therefore it is more likely that exceptions that are raised externally will be propagated, where necessary, following the process hierarchy.

Finally we would like to point out that the mechanisms described so far are sufficient to cope with external malfunctioning since this either manifests itself by a failure to reach an expected state within the expected time, thus giving rise to a `time_out` exception in the controlling process, or this may be explicitly signalled (as e.g. by an emergency-stop signal). The internal reaction to such signals may be delegated to separate processes, that —under normal circumstances— remain blocked. Therefore the 'scanning' of these signals can always be programmed as an add-on, and does not disrupt the basic system structure.

6 System structure

In section 2 we mentioned the failure of PLC-programs to tame the complexity of the control structure as the complexity of the controlled system increases. Decomposition of systems into parallel processes does have the attractive property that it prevents the complexity of the overall controlling system from exploding.
One example might suffice:
We take a section of a production line where at a certain point two components A and B enter the line, which subsequently undergo some operations and then are combined into a more finished product C. From a structuring point of view, we may well consider this a black box process with three bidirectional communications (Figure 2):
However the actual implementation of this control process may well consist of a

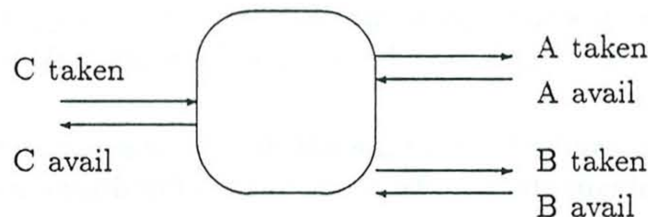

Figure 2: Structure model of a (sub)system, producing C out of A and B

further decomposition into subprocesses, depending on the potential parallellism present in that part of the production process. The three bidirectional links may thus eventually turn out to be links derived from internal processes.

What the example shows is the suitability of the parallel process model for scaling, and —as a corollary— for step-wise decomposition.

7 The Design of Industrial Control Systems

Because this lecture was primarily devoted to real-time control, we have postponed the subject that should come first. If we look at the problem of designing a control system from scratch, we will normally find a number of phases that represent the development of the system, say:

- specification
- global design ... detailed design
- implementation and installation
- testing

- operation

In the design phase both the decomposition into processes, and the definition of the signals to be exchanged between external world and control system take place.

In contrast to normal programs, testing can be a very expensive activity. On the one hand since it may bring to light certain defects in the design of the hardware, and on the other, because any unsatisfactory behaviour of the control system may have gross consequences for the devices being controlled. It is therefore quite attractive if a major part of testing can be done without connection of the controlled hardware. This form of testing, which not only may give clues to the correctness of the design, but also with regard to its performance is commonly called simulation.

One of the attractive features of the process interaction model for real-time control is that it is equally applicable in a simulation model. Yes, the simulation model might provide insights for the details of the eventual implementation.

The step from a ROSKIT-control system to a simulation model (or rather the other way around) is a small one. Additional processes may be written that model the real-time behaviour of the external world (e.g. the speed at which a particular component operates determines the time at which it will turn on a particular signal), and the time process of ROSKIT must be replaced by a scheduling algorithm based on a time-ordering of events.

These functions can be found in the S84 simulation package [11] [1], which has proved to be a very valuable tool in the development of real-time control systems.

8 Layers of control in industrial systems.

The real-time control of industrial systems is often called the process control. If we take a more encompassing look at industrial systems we can distinguish a component that is commonly called factory control. This component (or level) is responsible for the requests for production actions (eventually in reply to a request by customers) and keeps track of the results, e.g. it counts produced items, logs machine-time, mechanical faults, parameter settings and so on. These data may be entered manually, but most of them are directly derivable from the production process. It seems therefore natural to link both levels of control, in much the same way as the processes in real-time control, i.e. by the exchange of signals (messages) and suitable synchronization. The difference being the time scale, or response-times involved. Such interactions need not disturb the real-time behaviour of the controlled system since they can take place at a low priority. Now on top of the factory control level we can again implant another level of control, viz. the administrative level, which is responsible for producing invoices, orders, payment notices etc. The eventual control model arising for a factory is thus a fully integrated set of co-operating sequential processes, in a two dimensional array, the different rows representing the different control levels, and the control flow in each level represents the flow of items through the production process, be they material, conceptual, financial.

At each level we can normally recognize a bidirectional flow, where the counterflow can be viewed as the flow of acknowledges. This bidirectional flow is a well known

[1] A condensed description of both ROSKIT and S84 can be found in [10].

technique in administrative processes, but it is interesting to note that it is vital in the total control structure as well. As an example consider parts being removed from storage. This flow of parts is counteracted by the flow of signals that keep track of the number of parts in store, eventually causing resupply!

Each of these flows can model reality closely because the physical occurrences responsible for state transitions can be interlinked vertically through the exchange of signals and data, providing the overall control of the industrial system with up-to-date information at any instant during its operation. This model has been elaborated fully in Hans Arentzen's thesis [1]. The illustration in figure 3 of a complete factory model is taken from that thesis.

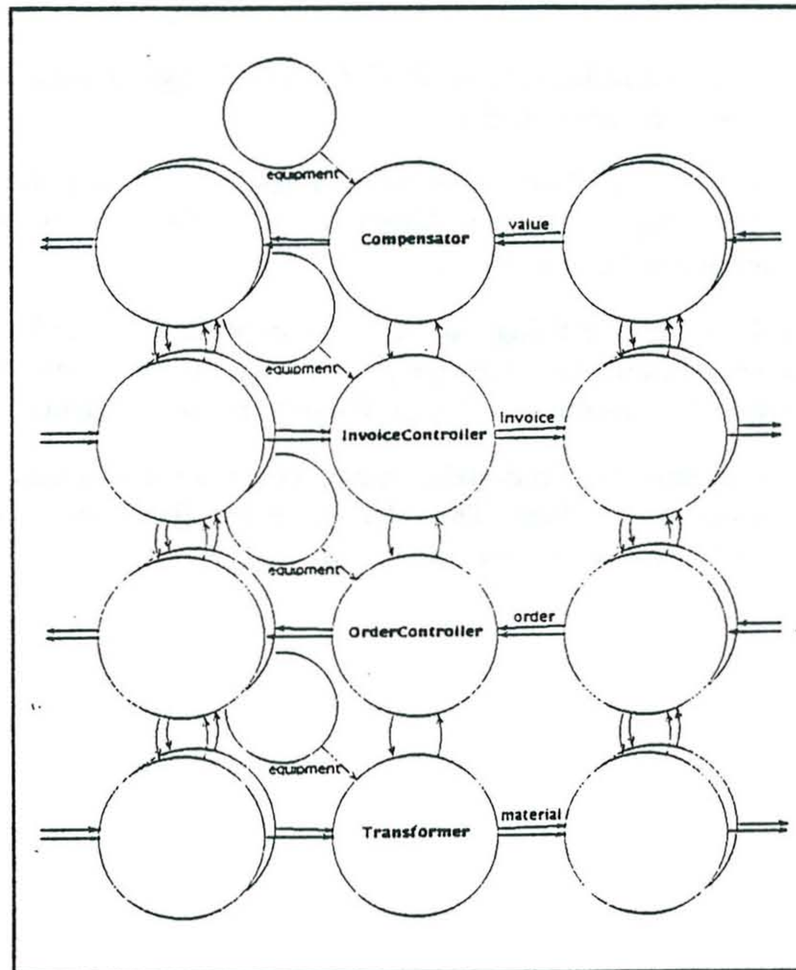

Figure 3: Factory model

9 Conclusions

A significant portion of industrial control applications can be implemented under the assumption that the controlling computer has ample capacity to meet the demands

of the physical processes being controlled. Under these assumptions the control structure is that of a set of sequential (but mutually parallel) processes, tightly synchronized with the external course of events and mutually synchronized as e.g. in Operating Systems. Control actions that have to take place at specific instants (i.e. in real-time) can easily be embedded in this general control structure. The ideas presented are by no means new (see e.g. [13]), but their value does not seem to be recognized by the majority of those responsible for the building of control applications. Several of the Vredestein plants in the Netherlands serve as a showcase for the success of control by means of parallel processes.

Acknowledgements
The author wishes to express his gratitude to those who have in fact developed the ideas presented in this lecture:

- Eelco Dijkstra and Herman Hebbink for conceiving c.q. implementing modules for an Operating System Kit.
- Koos Rooda for letting himself be inspired by the examples of synchronization in Operating Systems to formulate the basic concepts in simulating and implementing industrial control.
- Teunis Rossingh for being such a clever implementer, for letting always simplicity prevail, and for developing —virtually single handed— the control for the bicycle tire production line at Vredestein, Doetinchem.
- The management of Vredestein, in particular the former and present research managers at Doetinchem: Theo Boshuizen and Hans Arentzen, for putting so much confidence in Academic ideas.

References

[1] J.H. Arentzen,
Factory Control Architecture, Ph. D. Thesis,
Department of Mechanical Engineering, Technical University Eindoven (June 1989)

[2] T.P. Baker, A. Shaw,
The Cyclic Executive Model and Ada,
The Journal of Real-Time Systems **1**(1989), 7-25

[3] C. Bron,
Modules, Program Structures and the Structuring of Operating Sytems, in:
Trends in Information Processing Systems, 3d E.C.I. Conference, 135-153,
Lecture Notes in Computer Science 123, Springer-Verlag (1981)

[4] E.W. Dijkstra,
Hierarchical ordering of sequential processes, Acta Informatica **1**(1971), 115-138

[5] W. Halang,
Languages and Tools for the Graphical and Textual System Independent Programming of Programmable Logic Controllers, to appear in:
Microprocessing and Microprogramming(1989)

[6] H. Hebbink,
MOSKIT, A Modular Operating Systems Kit, M.Sc. Thesis,
Dept. of Math. & Computing Science, Univ. of Groningen (1985)

[7] International Electrotechnical Commission,
Working Draft*Standards for Programmable Controllers*, Part 3: Programming Languages,
IEC 65A(Secretariat) 90-I, December 1988

[8] L. MacLaren,
Evolving Toward Ada in Real Time Systems,
ACM SIGPLAN Notices **15**(Nov. 1980), 146-155

[9] C. Bron, E.J. Dijkstra,
Report on the Programming Language Modular Pascal (3d Ed.),
Department of Computing Science, Univ. of Groningen

[10] R. Overwater,
Processes and Interactions, Ph. D. Thesis,
Technical University of Eindhoven(1987)

[11] J.E. Rooda, S. Joosten, T.J. Rossingh, R. Smedinga,
S84 User Manual,
Dept. of Mech. Engineering, Univ. of Twente(1984)

[12] T.J. Rossingh,
ROSKIT, A Real-Time Operating System Kit,
Dept. of Mech. Engineering, University of Twente (January 1985)

[13] N. Wirth,
Toward a Discipline of Real-Time Programming,
Comm. A.C.M. **20**(Aug. 1977)

DISCUSSION

Rapporteur: Alan Tully

Professor Bron made the assertion that real-time systems may be designed using existing techniques which are well understood. He described the control of a bicycle tyre production line as an example, stating that it had been designed without the need for new methods or formalisms. Professor Mok agreed but pointed out that in the example given, the computer system was greatly underloaded. This view was echoed by several speakers. Professor Mok went on to say that in a more heavily loaded system with a reduced granularity on timing constraints, current techniques are inadequate and that further research is indeed required.