Informatics and Mathematical Modelling
Technical University of Denmark
Building 321
DK–2800 Lyngby
Denmark

HHL 97–12–20

Hans Henrik Løvengreen:

Design of Reactive Programs

Version 1.2

Summary

In this note, an approach to design of reactive programs is presented. The method is based on the *events* that occur between the program (system) and its environment. The events are temporally ordered in order to identify the main processes of the system. Finally, a design consisting of an abstract program is produced. This design may then be analyzed before a final implementation is made.

Contents

1 Introduction **1**

2 System identification **1**
2.1 Documentation 3

3 System structuring **4**
3.1 Event sequences 4
3.2 Finding event sequences 7

4 Program Structuring **8**
4.1 An abstract programming language 8
4.2 Transformation into abstract programs 10
4.3 Adding data flow 11

1 Introduction

A *reactive program* responds to external stimuli over time. Such programs typically appear in embedded systems monitoring and controlling a number of external physical devices.

Reactive systems are often implemented as concurrent programs. The design method presented in this note aims at reaching an abstract design consisting of a number of components interacting with each other. The components may be active ones representing processes or passive ones representing shared data stores (monitors).

Given an informal requirements specification, the method defines three steps in order to reach a design:

1. **System identification** The program system is identified by finding the *events* linking the system and its environment. These events by definition consitute the interface between the system and its environment. The events may be divided into *input* and *output* events. Also events are *typed* to indicate the information they carry.

2. **System structuring** In this step, the behaviour of the system is characterized. This is done by identifying temporal orders among the system events. The orderings are described in a terse textual notation. This results in a number of *event sequences* each corresponding to a sequential flow of events. The behaviour expressions are by default put in parallel at the outermost system level, and thus they identify the processes of the system.

3. **Program structuring** In this step, the event sequences are refined into (CSP-like) *abstract programs*. During this process, the information flow in the system is determined, and possible monitor components are identified.

Having reached a design, it is now possible to analyze the system. This may include its functionality (does it work correctly?) or its implementability. Such analysis is outside the scope of this note.

Implementation of the abstract model is not yet part of this note. Given a basic knowledge concurrent programming techniques, it should cause no problems to reach an implementation in a current programming language, eg. Java or Ada.

Often embedded systems involve real-time requirements to be satisfied. In this note, it is assumed that the program takes no significant time to react. A further analysis of schedulability is out of the scope of this note.

The method presented in this note is largely based upon ideas developed by A.P. Ravn, Hans Rischel and Benny Mortensen [RMR87].

2 System identification

By a system, we generally understand a structured collection of entities of a given universe to be considered as a whole. The system boundary divides the universe into thoses entities which are part of the system, and those belonging to the rest of the universe, the *environment* of the system. The system entities are often called *components*.

In the first step of the method, the system boundary is found by identifying the (interface) *events* that take place between the system and its environment.

An *event* is the abstraction of an interaction, ie. an activity involving more than one component. By modelling the interaction as an event, we consider the activity to be atomic: Either the activity results in an overall effect, or there is no effect at all.

For instance, the input of data from a user may involve setting up dialog boxes etc., but only when the user clicks the OK-button, the activity is considered an event.

Events are abstract entities and need not correspond to physical events like raising the voltage on a line or pressing a button.

Any *timing* period should be considered an interface event by it own. This way, the system will become *event driven*: the system will not act spontaneously, only as reaction to events.

The proper choice of events that suit the purpose of designing the system is often difficult. A guideline is to make events correspond to the activities that the system user would like to consider atomic.

A technique to find candidates for events is to mark the active verbs of the informal system requirements specification. These may then be analyzed to see whether they correspond to external activities, interface events, or internal activities.

The events may be partitioned into *input events* and *output events*. An input event carry information from the environment to the system and/or is (as least partly) under the control of the environment. An output event carry information from the system to the environment and is often under the total control of the system. The distinction between input and output may, however, not always be clear.

Data Types

Events are typed. Each event has an associated data type which represents the information exchanged during the event. Often events are simple signal not carrying other information than the fact that they take place.

Any well-defined type system could be used for typing events, such as the ML-type system, or types of specification languages such as VDM or Z.

Here we propose a traditional type system comprising simple types:

$NUMBER$	Some number.
$TOKEN$	Some unique code.
$\{\mathsf{id}_1, \ldots, \mathsf{id}_n\}$	Enumeration of distinct symbolic values.
$VOID$	Unit type with only one value: () .

and composite types:

$T_1 \times \ldots \times T_n$	Cartesian product (record, structure).
$T_1 \vert \ldots \vert T_n$	Union.
$T - LIST$	List of T-elements.
$T - SET$	Set of T-elements.

where the Ts are types. In this note, we shall not dwell upon issues such as type equivalence, subtyping etc.

2.1 Documentation

The interface events identified may be documented in an *event list* which for each event defines its direction (input or output), its data type (perhaps not yet defined), and its legend.

Futhermore, the relation of events to the environment may be shown in a *system diagram* showing the units in the environment and the events that link them to the system.

Example (Taximeter)

Consider the following informal specification of a simple taximeter:

> The taximeter can be switched on and off. When being switched on, the amount due shown on a display must be reset. While the taximeter is switched on, the amount should be incremented each minute as well as for each 100 meter signal received from the wheels. When switched off, the final amount remains on the display. When a "KM"-button is pressed, the total distance covered with the taximeter switched on must be shown for 10 seconds on a separate display.

We start by underlining the active verbs in order to find candidates for events:

> The taximeter can be switched on and off. When being switched on, the amount due shown on a display must be reset. While the taximeter is switched on, the amount should be incremented each minute as well as for each 100 meter signal received from the wheels. When switched off, the final amount remains on the display. When a "KM"-button is pressed, the total distance covered with the taximeter switched on must be shown for 10 seconds on a separate display.

We find that most of the marked verbs correspond directly to interface events. However, the reset and incrementation of the amount are internal acitivities whose result can be presented to the user by a common *show* event. Furthermore, since we assume that the display will not change if not asked to, the requirement that the amount *remains*, can be accomplished by lack of show events.

The events are naturally divided into input and output events as follows:

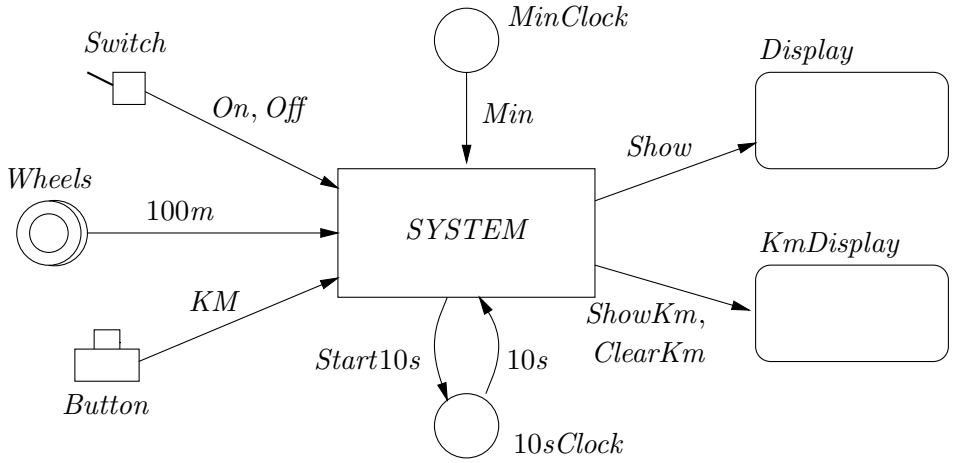
EVENT LIST:

Input event	**Data type**	**Legend**
On	*VOID*	Taximeter is switched on.
Off	*VOID*	Taximeter is switched off.
100m	*VOID*	Signal for each 100 meter.
KM	*VOID*	KM-button pressed.
Min	*VOID*	Regular signal from minute-clock.
10s	*VOID*	Signal 10 sec. after $Start10sec$.

Output event	Data type	Legend
Show	*NUMBER*	Show number on amount display.
ShowKm	*NUMBER*	Show number on distance display.
ClearKm	*VOID*	Clears distance display.
Start10s	*VOID*	Request for 10 sec. signal.

We assume that there is a minute-clock that ticks with regularly with a one minute period. If it is desired to synchronize this clock, a *StartMin* signal could be introduced. For the 10-second clock, we presume that it will signal once 10 seconds after $Start10s$.

Finally, we present a system diagram for the taximeter:



3 System structuring

In order to determine the system structure, the events are partially ordered according to the sequence in which they should occur in any system behaviour. For this, we use *event sequences* which characterize independent sequential behaviours of the system. Each independent event sequence gives rise to a major process of the program.

3.1 Event sequences

The syntax of event sequence expressions is given by:

$$s \quad ::= \quad e \mid \textbf{int} \mid \texttt{[}\, s \,\texttt{]} \mid s^* \mid s_1; s_2 \mid s_1 \,[]\, s_2 \mid s_1 \parallel s_2$$

Here e represents an event, and each s represents an event sequence.

An event sequence expression describes a behaviour according to the following informal semantics:

e *Event.* Represents the occurrence of an event e.

int *Internal.* Represents the occurrrence of an anonymous internal event.

$[\, s \,]$ *Option.* May behave like s or may internally choose to skip s.

s^* *Repetition.* Behaves as s any number of times. May terminate at the start of each iteration.

$s_1; s_2$ *Sequential composition.* Behaves first like s_1 and then like s_2.

$s_1 \,[]\, s_2$ *Choice.* Behaves like s_1 or s_2 depending on the first events of these.

$s_1 \parallel s_2$ *Parallel composition.* Behaves like s_1 and s_2 in parallel. That is, the behaviour of s_1 takes place concurrently with that of s_2 except for common events of s_1 and s_2 which must be synchronized. Terminates when both s_1 and s_2 have terminated.

Event sequences may be given names using definitions of the form:

$$name \;=\; s$$

Recursively defined event sequences are permitted and may be used in cases where the structure imposed by repetition may be too restrictive.

Syntax

The above grammer defines the abstract structure of event sequence expressions. When writing down such expressions, parentheses are used to resolve the ambiguity. However, in order to avoid lots of parentheses, we adopt the convention that the operators binds in the following order:

$$^* \;\; ; \;\; [] \;\; \parallel$$

where * binds strongest. Eg. the expression:

$$A;\; B^*;\; C \,[]\, C;\; D \parallel E$$

is understood as

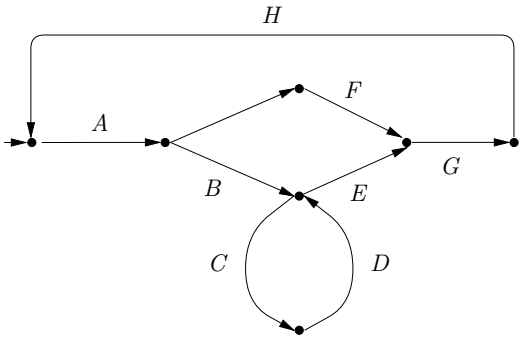
$$((A;\; (B)^*;\; C) \,[]\, (C; D)) \parallel E$$

Semantics

As semantic model for event sequences, we use a kind of automata or transition system. An *event transition graph* is directed graph whose edges may be labelled with events. Unlabelled edges correspond to internal events. Each node of the graph correspond to a state of the system. One of the nodes is marked as the *initial state*. The transition graph represents an abstract device (or machine) which may execute by performing events in cooperation with its environment or by performing internal events by itself.

In any state represented by a certain node, the system is willing to engage in any event labelling an outgoing edge. Such events are said to be *enabled*. We may think of the machine as having a button for each event. When an event is enabled, the corresponding button lights up. Now the environment may select and press a lit button. Hereby the machine will move along the edge labelled by that event. If there are several edges labelled with the event, one of them will be chosen by the machine. Now the machine has entered a new state, and a new set of events will be enabled. If a state has one or more unlabelled outgoing edges, the machine has the right to, but no obligation to, chose one of these by itself.

As an example of an event transition graph consider the following:

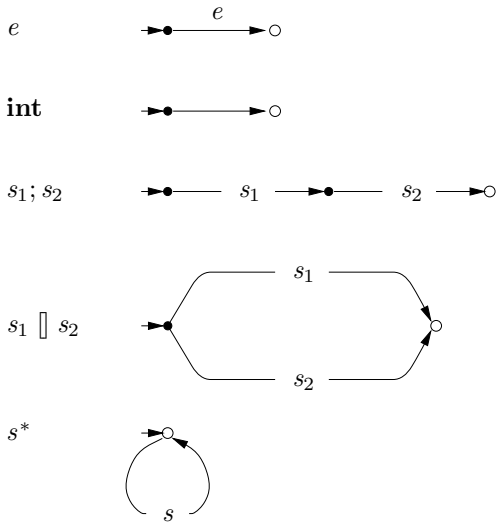


The initial state is marked by an arrow. This graph describes a machine that will initially perform the event A. After that, it may engage in B, or it may internally decide to abondon B in which case it will perform F. If, on the other hand, B is taken, it will be willing to engage in any number of C followed by D sequences. However, inbetween these, also the event E is enabled, and if the environment choses this, only the event G followed by H are possible, after which the machine is back in its initial state.

We would like this event transition graph to be the meaning of the event sequence

$$(A;\ (B;\ (C;\ D)^*;\ E \,[]\, \mathbf{int};\ F);\ G;\ H)^*$$

In general, we would like to define the semantics of an event sequence by generating a transition graph. To do this, we need to combine such graphs. For this purpose, it should be possible to mark nodes in transition graphs as *terminated*, indicating a state where the current behaviour may end. Here we indicate terminated nodes by small open circles. Furhtermore, let $-s\ \rightarrow$ denote the graph generated for s where $-$ represent all the edges leaving the initial state, and $\rightarrow$ represent all the edges leading to the final node. Now, the graph corresponding to an event sequence can be defined as follows:



The graph of $[\![\, s \,]\!]$ is given by the definition:

$$[\![\, s \,]\!] \triangleq (s \,[]\, \mathbf{int})$$

The graph of $s_1 \parallel s_2$ is given by interleaving of the graphs of s_1 and s_2. Normally, however, we are only interested in the graphs of (almost) sequential event sequences.

From the graph semantics we see that a repetition of, say, an event A can be *terminated externally* by another event B expressed by $A^*; B$ or *terminated internally* (eg. modelling a counter reaching some value) expressed by $A^*; \mathbf{int}$.

3.2 Finding event sequences

When ordering the events of a system, we aim at having a number of sequential processes running in parallel at the outermost level since this will give the most effective implementation in most langauges. Thus, we initially try to identify a number of (largely) independent sequential event orderings.

It is important to stress that the event sequences should describe the *meaningful dialogues* between the system and the environment. This means that events should appear only in situations where they make sense even though they may occur physically at other times.

A technique to find such a set of independent behaviour expressions is to proceed as follows:

1. First, only the *input events* are considered. The inputs events may be compared pairwise and it may be determined whether they are should appear sequentially (;), as alternatives ([]), together in some order ($\|$), or whether they are (largely) independent. In the latter case, they are put in separate event sequences.
2. Repetition and perhaps internal events are added to the event sequences.
3. Finally the output events can be inserted after the input events they are caused by.

This results in a number of *independent event sequences* which may at last be combined in parallel to make up the total system.

To be honest, the independent event sequence expression may sometime be only "almost sequential". The reason being that when adding the output events, we often find that two of these may occur in either order. Rather than making an arbitrary choice, we may put them in parallel and leave it to the implementation phase to find a proper order.

Sometimes, two (or more) event sequences should synchronize by participating in a common internal activity. In order to reflect this, new *named internal events* may be introduced and used in several event sequences.

Example (Taximeter continued)

> Looking at the input events we find that *Off* should follow *On*. Furthermore, the *Min* event and $100m$ only make sense, when the taximeter is switched on, ie. inbetween *On* and *Off*. Furthermore, *Min* and $100m$ may occur at the same places with the same effect, and thus they are naturally seen as alternatives. The *KM* event may occur at any time, but the $10s$ signal only makes sense when showing the distance. Adding repetition, this leaves us with two independent event expressions:

$(On; (Min \;[]\; 100m)^*;\ Off)^*$
$(KM;\ 10s)^*$

Adding the output events and naming the independent sequences, we get:

$$\begin{array}{lcl} count & = & (On;\ Show;\ ((Min \;[]\; 100m);\ Show)^*;\ Off)^* \\ inform & = & (KM;\ (Start10s \parallel ShowKm);\ 10s;\ Clear)^* \end{array}$$

Finally, the two expressions are combined in parallel:

$$taximeter \quad = \quad count \parallel inform$$

Thus, the taximeter should comprise two concurrent processes.

4 Program Structuring

The purpose of this design step is first of all to identify the dataflow in the system and to introduce the data stores needed. These may either be local to a proces taking the form of usual variables, or they may be shared between processes.

The result of this analysis is documented by an *abstract program* written in a general process language. Here we choose a CSP-like language where processes communicate over synchronous channels corresponding to the system events. The processes do not share variables. Shared data components are represented by server-like processes called *state monitors*.

The rationale of using CSP is that this model clearly separates local computation from interchange of information. The model may readily be implemented in both monitor-based as well as communication-based programming languages.

4.1 An abstract programming language

Our CSP-like proces langauge has the following syntax:

$$\begin{array}{lcl} p & ::= & a \mid \textbf{skip} \mid g \rightarrow p \mid \textbf{do}\ p\ \textbf{od} \mid \textbf{exit} \mid \\ & & p_1; p_2 \mid p_1 \;[]\; p_2 \mid p_1 \parallel p_2 \mid \\ & & \textbf{var}\ x : T \mid \textbf{chan}\ c : T \\ a & ::= & x := e \mid c\,!\,e \mid c\,?\,x \\ g & ::= & b \mid c\,!\,e \mid c\,?\,x \mid g_1, g_2 \end{array}$$

Here p is a proces, a an action, g a guard, c a channel, x a variable, e an expression, b a boolean expression and T a type.

Processes

Processes performs (atomic) actions. The behaviour of a process expression is informally given by:

a	*Action.* Process which performs the action a and then terminates.
skip	*Skip.* Process that cannot perform any action and just terminates.
$g \rightarrow p$	*Guarded process.* Process which which first passes the guard g and then behaves like the process p. The guard may declare variables whose scope is p.
do p **od**	*Loop.* Process which repeatedly performs p. May be terminated by execution af an **exit** in p.
exit	*Exit.* Exits the innermost enclosing loop.
$p_1; p_2$	*Sequential composition.* Process which first behaves like p_1. When p_1 terminates, it behaves like p_2.
$p_1 \,[]\, p_2$	*Choice.* Process which may behave like p_1 or p_2, depending on the first actions of these. If these are communications, the choice is determined by the current communication capabilities.
$p_1 \parallel p_2$	*Parallel composition.* Process which behaves like p_1 in parallel with p_2. Terminates, when both p_1 and p_2 terminate. Matching communications between p_1 and p_2 are synchronized.
var $v : T$	*Variable declaration.* Declares a variable x of type T. The scope of x is defined to be the rest of the (sequentially composed) process list in which the declaration occurs.
chan $c : T$	*Channel declaration.* Declares a channel c over which values of type T may be communicated. The scope of c is defined to be the rest of the (sequentially composed) process list in which the declaration occurs.

A process which is constructed without parallel composition is called a *sequential process.*

Actions

An action is an atomic activity. An action may either be internal or it may generate a communication event.

$x := e$	*Assignment.* Evaluates the value of e and assigns it to x.
$c\,!\,e$	*Output.* Evaluates e and outputs it on channel c.
$c\,?\,x$	*Input.* Inputs a value from channel c and assigns it to the variable x. The variable x is implicitly declared by the construct.

The communication is *synchronous* (as in CSP [Hoa78]), that is, an output action $c\,!\,e$ will wait until a matching input action $c\,?\,x$ can take place in a concurrent process.

Guards

A guard is an condition which must be passed before the process can proceed. A guard may be a boolean guard, an input/output guard, or a combination of these. All parts of a guard must be passed atomically for the guard to be passed. A boolean guard is passable only if it evaluates to *true*. An input/output guard is passed by performing the communication.

Note: The syntax and semantics allow for several input/outputs to be passed atomically, but this may be hard to implement.

Types and expressions

For types we may use the same types as for events, but maybe extended with more concrete types such as *Int* or *Real*. We asssume the existence of a standard (mathematical) notation for expressions, that expressions are well-typed, and that no expression evaluation will fail.

Definitions

As for event sequence expressions, we may define names for processes and use these. Futhermore processes may be recursively defined.

Conventions

We introduce a few convention in order to facilitate writing and reading of abstract programs:

- As for event sequence expressions, we introduce the convention that the operators bind in the following order (strongest first):

  $$; \quad \rightarrow \quad [] \quad \|$$

  When this is not appropiate, we use parentheses or square brackets to indicate the intended structure. Since ;, $[]$, $\|$ are associative, we may consider these to apply to lists of processes.

- If the type of a channel is *VOID*, input and output are just written as $c\,?$ and $c\,!$.

- Variables may be initialized: **var** $x : T := e$ corresponding to **var** $x : T;\ x := e$.

4.2 Transformation into abstract programs

As can be seen, the syntax of abstract programs resemble that of event sequences. In fact, each independent event sequence can be transformed into an *abstract program skeleton* by the following modfications:

- Events are replaced by communications. For each event e, a channel with the same name and type as the event is implicitly declared. Now, each input event I is is refined into an input action:

  $$I\,?\,x$$

  and each output event O is refined into an output action:

  $$O\,!\,e$$

- Seqential composition of events carry over to processes.
- Likewise for choices. However, the presence of a choice (typically between input events) is usually emphasized by using the communications as guards. For instance, $I_1; \ldots \,[]\, I_2; \ldots$ is transformed into:

$$[I_1 \,?\, x_1 \rightarrow \ldots \,[]\, I_2 \,?\, x_2 \rightarrow \ldots]$$

- Repetion s^* is generally replaced by a loop:

$$\textbf{do}\ p_s\ \textbf{od}$$

where p_s is the transformation of s.

However, whereas termination of s^* may occur implicly at the start of each iteration, termination of the **do** loop must be programmed explicitly. If a repetition is terminated by an event (typically an input event) this event must be considered as a possibility at the start of each iteration. If, for instance, the event sequence is $(I_1; \ldots)^*; I_2$, a choice between I_1 and I_2 must be made at the start of each iteration. If I_2 occurs, the loop is left:

do
  $I_1 \,?\, x \rightarrow \ldots$
[]
  $I_2 \,?\, x \rightarrow$ **exit**
od

If the repetition is terminated by an internal event, this must be transformed into a concrete exit condition. For instance, $(I_1; \ldots)^*;$ **int** should be transformed into:

do
  $I_1 \,?\, x \rightarrow \ldots$
[]
  $b \rightarrow$ **exit**
od

where b is some boolean termination condition.

- Parallel composition of output events within an event sequence may be resolved into a specific order.
- Outermost parallel composition of event sequences carry over to the corresponding processes.

4.3 Adding data flow

Having generated the abstract program skeleton processes, we now look for data sources for output data. In general, the data needed by an output $O\,!$ in some process may have three sources.

- The data may be found as a function f of a recently received data value still present in a local variable x. In this case, the output takes the form $O\,!\,f(x)$.

- The data may be computed from the history of the process itself. In that case, a process-wide variable is introduced, updated in connection with the relevant events, and finally used in the output action.
- The data may be computed using the history of one or more other processes. If so, a *state monitor* to hold the relevant information produced by the other process(es) is introduced. A state monitor is a passive process which may be operated upon by communicating with it over a set of internal channels.

If the output data still cannot be calculated, the system simply receives too little information to generate that output! Either more input events must be added, or the events must be processed in more situations.

Structure diagram

In order to illustrate the structure as well as the data flow in the system, a *structure diagram* may be drawn. The diagram should show the internal structure of the program, ie. the processes, the channels, the state monitors, and the external units. In the diagrams, we use circles to represent processes, rectangles to represent state monitors, hexagons to represent external units and arrows to represent internal and external channels.

Example (Taximeter contd.)

First, the abstract process skeletons are derived from the event sequences. We get two processes named after the corresponding event sequences. For the *Count* process, we should note that in each iteration, there are three possible next events: *Min*, $100m$ and the terminating *Off*. This can be programmed as shown:

```
Count =
  do
    On ?;
    Show !...;
    do
      Off ? → exit
    []
      [Min ? → skip [] 100m ? → skip];
      Show !...
    od
  od
```

The program illustrates the possibility of having nested choices. The transformation of *Inform* is straightforward:

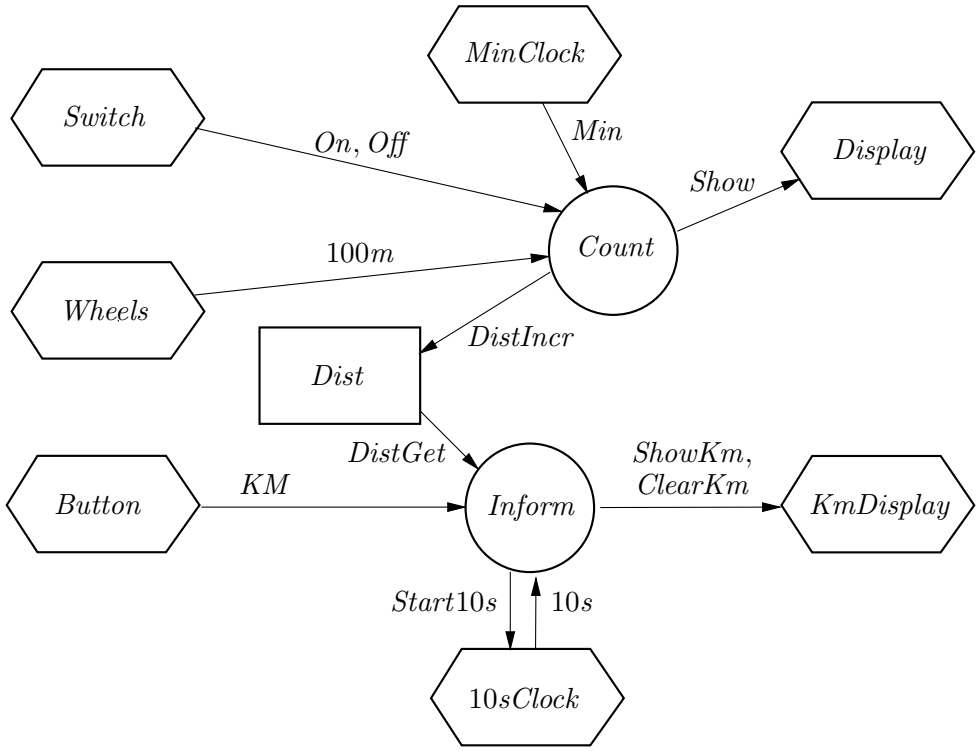
```
Inform =
  do
    KM ? ;
    ShowKm ! . . . ;
    Start10s ! ;
    10s ? ;
  od
```

Now we look at the data flow. We see that we need to deliver output to *Show* and *ShowKm*. For *Show* we find that the amount can be calculated on basis of the events occurring in *Count*. Thus, we can make do with a local variable of the *Count* process accumulating the amount.

For *ShowKm* we find that it cannot be calculated on basis of the events ocurring in *Inform* itself. But, luckily, it can be calculated from the events processed by *Count*, since it precisely processes $100m$ signals when the taximeter is switched on.

Thus the *Count* proces has the information needed by *Inform*. Rather than having them communicate directly, we introduce an intermediate state monitor *Dist* which can update and deliver the total distance covered with the taximeter on. Communication with the state monitor can be done through channel *DistIncr* which should make it increment the distance, and channel *DistGet* which should deliver the current distance.

This structure may be depicted in a structure diagram showing the processes, monitors and channels of the system:



ă

Now, the process skeleton may be completed by adding the local variables, and the communications with the state monitor:

```
Count =
  var amount : Real;
  do
    On ? ;
    amount := StartFee;
    Show ! amount;
    do
      Off ? → exit
    []
      [Min ? → skip [] 100m ? → DistIncr ! ];
      amount := amount + DeltaAmount;
      Show ! amount
    od
  od

Inform =
  do
    KM ? ;
    DistGet ? km;
    ShowKm ! km;
    Start10s ! ;
    10s ? ;
  od
```

The *Dist* monitor itself is readily implemented:

```
Dist =
  var km : Real := 0.0;
  do
    DistIncr ? → km := km + 0.1;
  []
    DistGet ! km → skip
  od
```

References

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8), August 1978.

[RMR87] Hans Rischel, Benny Graff Mortensen, and Anders P. Ravn. *Konstruktion af formålsbundne systemer.* Teknisk Forlag, 1987. (In Danish).