Combining signals and messages in computers

A. Richard
e-mail: androer@free.fr

Revised version 4/2008 with the participation of ***M. Peltier***
(e-mail: peltier@alumni.caltech.edu).
File available at: http://www.powys-lannion.net/Richard/Suggest.pdf
Former title: "A step toward automatic programming of computers"
Original title: "A suggestion to computer science amateurs"

Abstract

It is of the utmost importance that the requirements which a program is to satisfy should be rigorously expressed by a formal specification. Moreover, so that all interested parties can easily and efficiently co-operate in elaborating it, the specification itself should not depend on information technology. Simply by considering that all interactions and exchanges of information between the program and its environment, as well as between the computer and the external world, are equivalent to messages being received and emitted, primitives to combine messages are defined which allow the expression of any behaviour required from the computer executing the program. Although these primitives are independent of information processing and their use requires no knowledge of related techniques, it will be shown that they can be readily transposed into simple, well-known mechanisms or into combinations of such mechanisms. In this way, and as far as relations with environment are concerned, construction of an actual program may be fully automated given only its specification using these primitives.

Acknowledgements

All my thanks to Charles Derbez for his constructive criticism, and to Philip Shepheard for his assistance with the English version.

Contents

Chapter 1: Introduction

Thanks to the mathematicians and inventors B. Pascal, Ch. Babbage and J. von Neumann, we now have available programmable information processing machines which are capable of carrying out any task within their technical reach, just so long as the required programs can be produced.

The techniques of Software Engineering are used in industry to organise the work required and to verify the resulting products. What can be proposed to those users who dare write programs and have had enough of trial and error, especially of error, but cannot afford these techniques?

Without realising it, G. Boole opened up the possibility of a rational process for achieving the design and implementation of certain machines, starting from a formal specification of their required properties. This process only holds for a severely restricted set of machines, but extending it to complex mechanisms and to their implementation as computer programs appears to be within reach of a small team working towards that end. This paper describes the method for achieving such an extension.

As a simple example of a device to be specified, let us then consider a security system often found at the entrance to apartment blocks. In such systems, a keypad with ten numbered keys is used to control entry by way of a manipulation: certain keys must be pressed ***in sequence*** in a prescribed order.

If the required manipulation had been to press the keys ***simultaneously***, the function to be provided by the keypad system could have been described by a formal specification such as

```
[1001]      OPEN : C4 and C7 and C9 and C0
                      and not(C1 or C2 or C3 or C5 or C6 or C8)
```

which could then be considered as describing the interconnection of available hardware components implementing the Boolean operators **`and`**, **`or`** and **`not`**: a hardware implementation could then have been derived directly.

But how should the specification be written, and then used, to achieve a similar result in the case of the real keypad system? In the absence of any satisfactory answer, the designer, as in the past, is reduced to looking more or less randomly for a solution. That is, he imagines various possible ways of combining the available technical means, or abstractions of them, until he discovers one which appears to satisfy some informal specification.

On the other hand, the creation of a program —which is our main subject of interest— may be considered as proceeding logically through three phases, each of which produces a particular document:

1- the ***specification***, which defines the product by its behaviour, that is by relations of cause and effect; it is a description of the properties necessary for it to achieve the required function within its operating environment;

2- the ***model***, which describes an abstract machine designed to satisfy the specification, achieved by combining the abstract components of state machines;

3- the ***program***, which is the end result of adapting the model to computers; the program will then be transformed into a finished product by the compiler associated with a particular computer and operating system.

We will discuss below the basis of languages adapted to specifications and to models in order to allow for automation of the second and third phases in such a way as to eliminate any risk of error in carrying them out.

Chapter 2: Repetition and selection

Information is passed in various embodiments at the frontier of an information processing system and is more or less stable over time. All information exchanged at the frontier of the basic computer itself is however transient and in standard digital formats which we will refer to as signals or more generally as messages.

2.1 -Repetitive signals

Signals are the atoms of this transient information. A particular signal will act on a machine which has been designed for it: we will say that it has been ***perceived*** by the machine. On the other hand, a signal may result from the machine's operation: we will say that it is ***emitted*** by the machine.

A well-designed machine will not react to a signal which it has not been designed to perceive: there are no fortuitous signals in information processing. In the same way, the machine will only react by signals which it has been designed to emit. It can only perceive or emit signals which have been foreseen during the design of the machine's hardware and software.

The machine is constructed and programmed to react on many occasions in the same way to identical solicitations: the perception or emission of the signals to be considered is thus potentially ***repetitive***. Each relevant signal is made up of ***dispatches***, which may ***occur*** an unlimited number of times; a signal will be said to occur whenever one of its dispatches occurs. Timing signals may occur regularly, but repetition in itself does not imply any regularity. The name which may be given to a signal will not distinguish between different dispatches; however, it may be used to refer to a particular dispatch if there is no ambiguity.

Now let a ***stroke*** be the dispatch of any one of the signals perceived, or of several such signals simultaneously. It will sometimes be convenient to think of them as belonging to a signal `STROKE`, which can enter in formulations as well as `NEVER`, the signal which never occurs. Delays only limit the performance of a machine, and will be disregarded except when achieved by explicit use of periodic signals. Consequently, ***any dispatch of an output signal will of necessity occur in coincidence with a stroke***: it will be the result of a ***selection*** amongst the dispatches of the signals perceived.

The specification of the machine thus defines a ***global selection***, i.e. which dispatches of the input signals will be transmitted as dispatches of which output signals, and that for all relevant patterns of dispatches perceived.

Simple selections will be defined of which combinations will suffice to perform any global selection. A simple selection will ***produce*** a single signal as a ***result***, which will be made up of all the dispatches selected. It is identified by a symbol called a ***selector***, which appears in the expression of the result in addition to the identification of the messages submitted to selection, the ***selectands*** (or "seligends" for strict Latin scholars!).

Results exist as abstractions; their dispatches occur independently of their emission by a machine; the result of a selection may thus be used as a selectand without being emitted.

Signals produced by different selections or selection combinations will be said to be ***equivalent*** (symbol ==) if their dispatches always occur together.

2.2 -The selection `then` and its referential

This is the only selection to combine signal dispatches which are not simultaneous. Before it may be defined, the notion of ***immediate succession*** must first be discussed.

There can be no point in considering immediate succession in an absolute sense, taking into account all the signals of the Universe, without any means of knowing them, nor of observing their dispatches. It would also be absurd to take into account those signals at the machine's interface which have no bearing on a particular desired reaction. Immediate succession is thus relative to an explicit universe, a set of signals, the ***referential.*** Any such selection applies in particular to two of these signals: let them be `A` and `B`. By definition, a dispatch of `B` will be said to be in immediate succession to a preceding dispatch of `A` each time that no dispatch of any other constituent of the referential ***separates*** `A` and `B`. Dispatches simultaneous to those of `A` or `B` do not separate them. `A` is the ***initial*** signal, `B` is the ***terminal*** signal.

The selection **`then`** corresponds to this definition of immediate succession. The selectands are the constituents of the referential; initial and terminal signals are the ***succession selectands***. A dispatch of the result occurs, or more briefly, the succession occurs, ***each time the immediate succession takes place, at the stroke at which it is completed.*** At any given time, only past and present dispatches have been perceived, so that the result may only occur upon a dispatch of the terminal signal.

Let a ***mark*** be the dispatch of any selectand. After a mark at which the initial signal occurs, ***the succession will occur whenever the terminal signal occurs at the next mark.***

Consider for example a referential of three signals `A`, `B` and `C`. There are nine first order successions of two dispatches:

```
1)    A  A    A then A among {A,B,C}    A then A heed {B,C}
2)    A  B    A then B among {A,B,C}    A then B heed C
3)    A  C    A then C among {A,B,C}    A then C heed B
4)    B  A    B then A among {A,B,C}    B then A heed C
5)    B  B    B then B among {A,B,C}    B then B heed {A,C}
6)    B  C    B then C among {A,B,C}    B then C heed A
7)    C  A    C then A among {A,B,C}    C then A heed B
8)    C  B    C then B among {A,B,C}    C then B heed A
9)    C  C    C then C among {A,B,C}    C then C heed {A,B}
```

Two expressions of the corresponding result are shown in each case: a ***full expression***, where all constituents of the referential appear after **`among`**; and, to the right, a ***concise expression***, where the ***inhibitor(s)***, i.e. the selectand(s) whose dispatches can break the succession, appear after **`heed`**. The symbols **`then`**, **`among`** and **`heed`** are selectors.

The following illustration shows hypothetical dispatches of the three selectands and of an extraneous signal `E` (ineffective), together with those dispatches of the terminal signals which make up the results of the selections. Each dispatch is indicated by "`*`"; columns correspond to strokes.

```
  A     *         * *   *        *        *   *
  B       *    *  *       *  *  *      *  *   * *
  C          *    *  *             *   *  *   *
  E         *            *    *       **
marks   * *  * *  * **  * *  *  ** *   *  *   *

 A A                *                         *
 A B      *               *                   * *
 A C                 *             *          *
 B A              * *            *        *   *
 B B              *          *  *         *   * *
 B C         *    *                       *   *
 C A                *   *                 *   *
 C B           *                       *  *   * *
 C C                                   *  *   *
```

Dispatches of different results may occur simultaneously; they all coincide each time the three selectands occur together twice running.

Immediate succession and the selection **then** may be extended to several or indeed to all the constituents of a referential, and to any number of dispatches; however many elements there may be in the succession, dispatches of the result will be selected amongst those of the terminal signal.

```
                B then B then B heed {A,C}
```

is an example of ***second order*** **then** selection. Its result occurs whenever `B` occurs for the third time in immediate succession, with the referential `{A,B,C}`. If the set of inhibitors is empty, **heed** and the braces may be omitted in the expression of the result. For example, the third order selection

```
                A then A then B then C
```

is an expression of the signal which occurs at every stroke at which the immediate succession of dispatches of `A`, then `A`, then `B`, then `C` is completed, with `{A,B,C}` as referential.

The concise form suggests that the same result could be achieved by first considering the successions defined on a reduced referential consisting only of the succession selectands, and then eliminating those broken by dispatches of the inhibitors. It is thus clear that one or more of the succession selectands may be included in the inhibitor set without impairing the result, since their dispatches belong to the sequence specified. The following equivalence thus holds:

```
           A then B heed {A,B,C}  ==  A then B heed C  ==  A then B among {A,B,C}
```

2.3 -Instantaneous selection of dispatches; binding strength of selectors

Dispatches of different input signals may be perceived simultaneously; moreover, dispatches of the results of **then** selections occur in coincidence with those of their terminal selectands. The resulting need for selecting amongst simultaneous dispatches will be satisfied by instantaneous selectors. As opposed to **then**, an instantaneous selection does not involve past dispatches. Its referential, always implicit, is constituted by its selectands. Three basic instantaneous selectors are defined.

• Dispatches selected by **when** are those of the first selectand which occur simultaneously with any dispatch of the second; either of the expressions

```
                S1 when S2
```

denotes a signal which occurs at each stroke at which both `S1` and `S2` occur. Especially

```
        [2301]      S when STROKE  ==  STROKE when S  ==  S
```

Since all the dispatches of the result of a **then** selection are simultaneous to dispatches of its terminal selectand, the following equivalence holds:

```
        [2302]      A then B heed {...}  ==  (A then B heed {...}) when B
```

• Dispatches selected by **whenno** are those of the first selectand which do not occur simultaneously with any of the second, so that:

```
                S1 whenno S2
```

denotes a signal which occurs at each stroke at which `S1` occurs and `S2` does not. The signal

```
                STROKE whenno S
```

occurs at each stroke at which `S` does not, but

```
                S whenno STROKE
```

never occurs.

• The expression

```
                S1 orelse S2
```

denotes a signal occurring at each stroke at which `S1` and/or `S2` occur(s). One may consider that dispatches selected by **orelse** are those of the first selectand and those of the second which are not simultaneous to any one of the first. Hence:

```
        [2303]      S1 orelse S2  ==  S1 orelse (S2 whenno S1)
                                  ==  S1 orelse S2 whenno S1
```

Since `NEVER` occurs at no stroke,

```
                NEVER orelse S  ==  S orelse NEVER  ==  S
```

The symbol **any** will be used, associated with braces, in an alternative notation for multiple **orelse** expressions:

```
[2304]      any {a,b...,z}  ==  a orelse b... orelse z
```

The convention adopted in signal expressions in order to reduce the need for brackets around subexpressions is that **when** and **whenno** have greater binding strength than **orelse**. Instantaneous selections of equal strength are carried out from left to right. All instantaneous selectors are weaker than **then** and **heed**.

2.4 -Messages and types

A ***message*** is a set of one or more signals, whose elements are its ***constituents***. A message occurs whenever one at least of its constituents occurs. A particular message can be defined by a bracketed list of valid signal expressions.

A signal is thus just a particular message, with a single constituent:

```
[2401]      {S}  ==  S
```

This implies that “signal or message” has the same meaning as “message”.

The ***token***

```
            any {S1...,Sn}
```

of the message `{S1...,Sn}` is the signal which occurs whenever any of its constituents occurs. Permutation or duplication of constituents is then neutral with respect to the token:

```
[2402]      any {A,B}  ==  any {B,A}  ==  any {A,B,A}
```

If there is only one constituent, the token is equivalent to it:

```
[2403]      any {S}  ==  S
```

As defined in 2.2, the referential and the inhibitor of, for example, the selection

```
            A then A heed {B,C}  ==  A then A among {A,B,C}
```

are messages. The marks of the selection are then the dispatches of the token of its referential

```
            any {A,B,C}
```

which will be referred to as its ***marker.***

Two messages are said to be ***similar*** if there exists a reciprocal correspondance between their constituents.

Two messages are said to be equivalent if they are similar and if corresponding constituents are equivalent. Equivalence between messages implies equivalent tokens.

As signals, messages may be given names.

A ***typed message*** is defined as ***carrying*** at each dispatch a ***value*** within the set of a specific type. Boolean, numerical and alphanumerical messages will be encountered frequently, but structured types are not to be excluded.

Such a message may be thought of as an extended signal capable of dispatches of various sorts; for example, if dispatches of a given output signal are flashes at some given location on a display, each dispatch of a given typed message will similarly be a flash at one amongst various given locations, each corresponding to a particular value within the type. From the theoretical point of view, such a message is thus considered as a collection of as many constituents as the type has possible values. These are exclusive signals, in as much as their dispatches cannot be simultaneous. Such a constituent will be referred to by the name of the message indexed by a value within the type or, if necessary, by the extra value `NIL`; for a message `Y` with values ranging from `v1` to `vn`, this gives

```
[2404]      Y  ==  {Y[v1]... ,Y[vn], Y[NIL]}
```

In the case of a Boolean message, two values should suffice:

```
[2405]      YB  ==  {YB[FALSE], YB[TRUE]}
```

whose token is

```
[2406]      any YB  ==  YB[FALSE] orelse YB[TRUE]
```

The ***contents*** of a message are the value it carries. A constant `K` may be considered as a message carrying the same value at every stroke; its token is `STROKE`:

```
[2407]      any K  ==  STROKE
```

Two messages of the same type are similar. They are equivalent if their tokens are equivalent and if their contents are equal at each dispatch.

A signal may be considered as a message carrying nothing, an ***empty message***.

2.5 -Selections applicable to both untyped and typed messages

If a message has been given a name `M`, its token may be denoted

```
            any M
```

where the symbol **`any`** will be considered as a selector applied to the constituents of `M`. In a multiple **`orelse`** selection of signals by **`any`**, the name of a message will be considered as standing for its token:

```
[2500]      any {...M...}  ==  any {...any M...}
```

The selectors **`when`** and **`whenno`** will be freely used with semantics derived from those defined in 2.3, by the two rules:

• the second message is replaced by its token:

```
[2501]      Ma when Mb  ==  Ma when any Mb
[2502]      Ma whenno Mb  ==  Ma whenno any Mb
```

• the selection **`when`** or **`whenno`** applies to each constituent of the first message.

Let Ma_1 to Ma_n be the constituents of `Ma`; these rules can bee formalised into the equivalences:

[2503] Ma **when** Mb == {Ma_1 **when any** Mb..., Ma_n **when any** Mb}

[2504] Ma **whenno** Mb == {Ma_1 **whenno any** Mb..., Ma_n **whenno any** Mb}

Hence, the result is similar (see 2.4) to `Ma`. Owing to this similarity, **`when`** and **`whenno`** conserve type, if any. The token of the result is the result of the selection of the tokens since, in both cases, the same dispatches of the constituents of `Ma` are selected:

```
[2505]      any (Ma when Mb)  ==  any Ma when any Mb
[2506]      any (Ma whenno Mb)  ==  any Ma whenno any Mb
```

The selection **`then`** provides a message from three messages `M1`, `M2` and `H`, which constitute a referential `{M1,M2,H}`, and whose marker is **`any`**`{M1,M2,H}` (see `[2500]`). The result is similar to the ***terminal message*** `M2`. The dispatches of its constituents are those of the corresponding constituents of `M2` which are in immediate succession to the token of `M1`:

```
[2507]      M1 then M2 heed H  ==  (any M1) then M2 heed H
```

where the brackets are not necessary. Let $M2_i$ be a constituent of `M2`; the corresponding constituent of the result has for expression:

any M1 **then** $M2_i$ **among** {M1,M2,H}

A new selector **`next`** is defined, which must be combined with **`when`** or with an instantaneous operator.

```
[2508]      M1 when next M2 heed H  ==  M1 when next any M2 heed H
```

is similar to `M1`. Its current constituent is:

$M1_i$ **then any** M2 **among** {M1,M2,H}

Following `[2500]`, replacing a message selectand by its token preserves the marks. In the cases of both the selections **`then`** and **`when next`**, an expression for the token of the result is thus obtained by replacing both messages by their tokens in the original expression:

```
[2509]      any (M1 then M2 heed H)  ==  any M1 then any M2 heed H
[2510]      any (M1 when next M2 heed H)  ==  any M1 then any M2 heed H
```

The selection **`orelse`** will apply to similar messages. If `M1` and `M2` are two such messages of corresponding current constituents $M1_i$ and $M2_i$, the corresponding constituent of the result has for expression:

`M1i` **orelse** `M2i` **whenno any** `M1`

Dispatches of the second message are only selected ***when no dispatch of the first occurs***. This is referred to as ***preference***. A dispatch of the result, which is similar to `M1` and `M2`, corresponds to each dispatch of one or the other selectand:

`[2511]` **any** `(M1` **orelse** `M2)` `==` **any** `M1` **orelse any** `M2`

An equivalence analogous to `[2303]` stems from the definition:

`[2512]` `M1` **orelse** `M2` `==` `M1` **orelse** `M2` **whenno** `M1`

2.6 -The case of typed messages

To be similar, two typed selectands `Y1` and `Y2` must be of a common type, which will also be that of the result of an **orelse** selection. The constituent with index `v` of the latter will have for expression:

`[2601]` `Y1[v]` **orelse** `Y2[v]` **whenno** `Y1`

The contents of a dispatch of the result are thus those of `Y1`, unless `Y2` occurs alone.

Some selections correspond to operations ***defined between the types*** of two messages `Y1` and `Y2`. If ***op*** is the symbol of such an operation, the result

`Y1` ***op*** `Y2`

is a third message which occurs at each stroke at which the first two occur simultaneously:

`[2602]` **any** `(Y1` ***op*** `Y2)` `==` **any** `Y1` **when any** `Y2`

When `Y1` and `Y2` carry the values `v1` and `v2`, the result carries the value of

`v1` ***op*** `v2`

In the case where `v1` ***op*** `v2` is not defined, the contents are `NIL`.

The result is also defined for a constant `k`, i.e. a stroke-independent value, and a typed message `Y`; its token is that of `Y`:

`[2603]` **any** `(k` ***op*** `Y)` `==` **any** `Y`

`[2604]` **any** `(Y` ***op*** `k)` `==` **any** `Y`

When `Y` carries the value `v`, the respective contents of the results are the values of:

`k` ***op*** `v`

`v` ***op*** `k`

Monadic operations ***mop*** defined on the type are applicable; the token remains unchanged. For a Boolean message,

`[2605]` ***mop*** `Y[v]` `==` `Y[`***mop*** `v]`

Between two typed messages, the selector **next** is always combined with **when** or with an instantaneous operator ***op*** compatible with both types:

`Y1` ***op*** **next** `Y2` **heed** `H`

occurs at each stroke at which a dispatch of `Y2` is in immediate succession to a dispatch of `Y1` with the referential `{Y1,Y2,H}`; it carries the product of the operation ***op*** between the contents of the preceding dispatch of `Y1` and the current contents of `Y2`. Replacing `Y1` and `Y2` by their tokens does not alter the marks, so that the following equivalence holds:

`[2606]` **any** `(Y1` ***op*** **next** `Y2` **heed** `H)` `==` **any** `Y1` **then any** `Y2` **heed** `H`

This selection may be extended to any number of messages. For the values `v0`, `v1` ..., `vn` carried respectively by the dispatches of `Y1`, `Y2` ..., `Yn` in immediate succession,

`Y0` ***op1*** **next** `Y1` **...** ***op*****n** **next** `Yn` **heed** `H`

occurs carrying the product from left to right:

`v0` ***op1*** `v1` **...** ***op***`n vn`

The full form (see 2.2) may be used for **then**, **when next** or ***op*** **next** selections.

A function `f` of an argument of a given type is also applicable to messages of this same type:

`[2607]` `f(Y[v])` `==` `Y[f(v)]`

A function of two or more arguments defines a selection amongst messages of the corresponding types. A dispatch of

```
                f(Y1, Y2, Y3)
```

occurs at each stroke at which `Y1`, `Y2` and `Y3` occur simultaneously; the token is then

```
                any Y1 when any Y2 when any Y3
```

For values `v1`, `v2`, `v3` carried by the arguments, the contents are

```
                f(v1, v2, v3)
```

The subject of such ***banal*** functions, computing only content values, will be examined in chapter 5.

Traditional arithmetic, Boolean, relational operators etc... will generally be found in message expressions; their binding strength will follow the usual rule. Overall, operators will be weaker than selectors; following previous indications, the hierarchy is thus (highest strength first):

any, **not**, arithmetic ***mop***'s
then, the couples **when next** and ***op*** **next**, **heed**, **among**
when, **whenno**
orelse
Boolean, arithmetic and relational ***op***'s as usual

Untyped messages will only be used as referentials or inhibitors or in other positions where they may be replaced by their tokens, i.e. as second selectands of **when** or **whenno**, or as limiters in prospecting expressions or prospectives (see chapter 6).

Chapter 3: Behavioural specifications

In this chapter, we will show how to provide a formal specification of the behaviour the final product must display. Such a specification must take into account all relevant cases, and must rigorously prescribe the desired actions of the computer in each case.

3.1 -A keypad entry system: behaviour formulation

Consider once more the keypad entry system discussed briefly in the Introduction. The signals `C0`...,`C9` from the keypad constitute the universe of the associated processing device. The signal which must be emitted to unlock the door can be obtained through the extended **then** selection with this universe as a referential; a dispatch of the result occurs as soon as and each time the keys have been pressed in the correct sequence for the chosen entry code, as informally specified.

The specification of the entry system's program will reduce to a single ***output signal definition***, giving a name to the only signal to be emitted, which is the result denoted by the right-hand side expression. For the code `4790`, this will be

```
   [3101]      OPEN : C4 then C7 then C9 then C0
                              heed {  C1,C2,C3,  C5,C6,  C8  }
```

which describes the required behaviour by a third order **then** selection.

The right-hand side of the definition may be said to be ***the expression of the signal named on the left-hand side***. No mechanism need be imagined in order to write this expression, so that it may indeed be written without any knowledge of information processing techniques.

What actually distinguishes the signals `Ci` from one another is that they are perceived at distinct points of passage, or ports, at the frontier of the machine.

The system specified by `[3101]` recognises a ***spatio-temporal pattern*** at the ten points `C0` to `C9` and over four strokes, as shown below:

```
stroke number  -3 -2 -1  0
           C0   .  .  .  !
           C1   .  .  .  .
           C2   .  .  .  .
           C3   .  .  .  .
           C4   !  .  .  .
           C5   .  .  .  .
           C6   .  .  .  .
           C7   .  !  .  .
           C8   .  .  .  .
           C9   .  .  !  .
```

Dispatches denoted "`!`" are those required to generate the output signal; those denoted "`.`" ***may also be perceived*** but have no effect with respect to the output signal.

The stroke numbered `0` is the ***moving time reference*** which passes from one stroke to the next at each dispatch of any of the signals perceived. At each stroke for which the above spatio-temporal pattern is recognised, a dispatch of the output signal occurs.

It must be pointed out that extra dispatches of input signals do not inhibit `OPEN` as defined by `[3101]` when they are simultaneous with dispatches in correct succession. In the present case however, the associated circuit can be designed to discriminate between very close keystrokes, even if intended to be simultaneous, and efficiently enough to avoid any significant increase in the risk of intrusion due to lucky typing, which is one in ten thousand for a four digit code.

Can any possible behaviour required be described in a similar way, without making any assumptions about the means for achieving it? We need only assume that the specification describes in fact spatio-temporal patterns of signals to be recognised over a finite number of strokes.

3.2 -Mutually exclusive successions of signals

Now consider a machine which perceives three signals `A`, `B`, `C`. There are seven possible single or simultaneous dispatches of them and seven corresponding mutually exclusive signals:

```
X1 : C whenno A whenno B
X2 : B whenno A whenno C
X3 : B whenno A when C
X4 : A whenno B whenno C
X5 : A whenno B when C
X6 : A when B whenno C
X7 : A when B when C
```

of which one and only one occurs at each stroke. In these definitions, the `Xi`'s are just names given to the messages denoted by corresponding right-hand side expressions.

Two successions will be ***exclusive*** one of the other if they never occur simultaneously. Two successions of the above signals `Xi` are exclusive if and only if they differ by at least one of their selectands; the signal `SXjk` corresponds to each pair `Xj`,`Xk`. For example, any two of:

```
SX14 : X1 then X4 heed {X2,X3,X5,X6,X7}
SX11 : X1 then X1 heed {X2,X3,X4,X5,X6,X7}
SX41 : X4 then X1 heed {X2,X3,X5,X6,X7}
```

are mutually exclusive.

Consider again the dispatches of `A`, `B`, `C` and `E` illustrated in 2.2 (`E` does not belong to the referential), and let us show the signals `SXjk`. Those which do not occur are not shown:

```
A          *         * *   *        *        *   *
B            *    *  *       *  *  *      *  *   * *
C               *    *  *             *   *  *   *
E              *            *    *       **
i          4 2  1 2  7 41  4 2  2  24 1   3  7   7 2
SX12                  *
SX13                                      *
SX14                       *
SX21                *
SX22                            *  *
SX24                                *
SX27                 *
SX37                                         *
SX41                    *             *
SX42         *               *
SX72                                               *
SX74                   *
SX77                                             *
```
At each stroke, a dispatch of one of `SXjk` occurs as the result of the recognition of a particular spatio-temporal pattern, except at the first. For `SX27`, this pattern is:

```
stroke number  -1  0
            A       !
            B    !  !
            C       !
```
Dispatches denoted “`!`” are required; no others are tolerated.

More generally, a succession of order n will occur on the last of n+1 successive strokes as a result of the pattern being recognised over these n+1 strokes; a dispatch of the corresponding signal occurs at this same stroke, and never before the stroke of rank n+1 after startup. The signal defined by

`SX1422 : X1 then X4 then X2 then X2 heed {X3,X5,X6,X7}`

occurs whenever a particular exclusive pattern of `A`, `B`, `C` is recognised over four strokes.

3.3 -Describing the behaviour of a machine

A specific port is allocated to each signal at the frontier of a machine, whose rôle thus consists in emitting dispatches at its output ports as a function of those perceived at its input ports. The desired behaviour can thus be defined by enumerating for each output port the relevant conditions in which a dispatch is required. Such a dispatch is emitted immediately upon the situation occurring.

Now each such situation can in fact only be the existence of a spatio-temporal pattern on the entry ports to the machine over a certain number of strokes, possibly only one, to which will correspond an exclusive signal defined either by an exclusive succession, or possibly by an instantaneous exclusive selection.

Any output signal will thus be the result (defined in 2.3) of the selection by **`orelse`** of as many exclusive signals as there are different cases requiring the emission of a dispatch. For example, the dispatches of the signal

`SX1422 orelse SX53 orelse X6`

derived from the signals defined in 3.2 correspond to cases involving either four, two or one stroke(s).

There will be as many definitions as there are output ports. This also holds for the behaviour of machines perceiving or emitting messages of any kind and for the programs conferring this behaviour upon them, since as we have seen, messages are collections of signals. The input or output port of a message will be considered to be the collection of ports of its constituents.

The aim of the preceding discussion is to show that the selections introduced above in 2.2 and 2.3 are sufficient to describe everything required in a specification, and not to restrict their use. In particular, **`then`** selections will not be restricted to instantaneous exclusive signals in defining output messages, and their referentials will usually be subsets of the machine's universe.

In addition to exclusive signals, ***intermediate messages*** implying no output ports will be freely defined by ***auxiliary definitions*** giving names to messages denoted by their right-hand parts. They will be distinguished in this document by names starting with "`1`". Any such transformation achieved solely by intermediate definitions provides a text strictly equivalent to the original specification.

A comment on the use of the full form in **`then`** and **`next`** selections may be appropriate at this point. Although this form requires more space than the concise form, an overall gain may be achieved when it allows for a separate definition of a referential common to several successions. Defining an explicit common referential in this way may also contribute to greater clarity in the specification itself. For example, the definitions of `SX14`, `SX11`, `SX41` and `SX1422` in 3.2 may be replaced by:

```
SX14 : X1 then X4 among 1RX |
SX11 : X1 then X1 among 1RX |
SX41 : X4 then X1 among 1RX |
SX1422 : X1 then X4 then X2 then X2 among 1RX |
1RX : {X1,X2,X3,X4,X5,X6,X7}
```

A specification is made up of message definitions, whose right parts are message expressions. A message expression is a notation for the result of a selection applied to selectands which may themselves be either message names, basic expressions or expressions for unnamed intermediary results. As a rule, a name won't be defined more than once in a given specification.

3.4 -Use of intermediate messages: specification of "mutual exclusion"

In a file system providing multiple time-shared read and write access, data coherence has to be preserved. This may be achieved at the record level by a "mutual exclusion" server which prevents read access to a record while it is being updated, and prevents write access while it is being read.

A specification for such a server will be given below. The server will basically delay access to a record when a request to read (write) occurs during writing (reading) of this record. To that end, the server must produce authorisations in response to the requests received.

Permission to access a record is requested by one of the two typed messages

`MReqWr` or `MReqRd`

which carry as their contents the alphanumerical key to the record concerned. The write or read operation can then only take place when the corresponding authorisation signal

`AutWr` or `AutRd`

is emitted by the server. The end of the operation must then be signalled to the server by emitting

`EndWr` or `EndRd`

Our intention here is now to develop a formal specification for the generation of the authorisations `AutWr` and `AutRd`; there are three cases of possible conflict:

1- a read request received while a write operation is ongoing (strictly speaking between `MReqWr` and `EndWr`) will produce the dispatch of

```
[3401]      1MXRd : MReqWr = next (MReqRd whenno EndWr) heed EndWr
```

simultaneous with `MReqRd`, and carrying the product of the comparison between record keys by the relational operator =; the value `TRUE` will be carried in case of conflict;

2- similarly, a write request occurring while a read operation is ongoing will produce the dispatch of

```
[3402]      1MXWr : MReqRd = next (MReqWr whenno EndRd) heed EndRd
```

simultaneous with MReqWr;

3- when both requests are simultaneous, the result of their instantaneous comparison is a dispatch of the intermediate signal:

```
[3403]       1MXsim : MReqRd = MReqWr
```

Two signals 1DelRd and 1DelWr define which request is to be delayed in case of conflict:

```
[3404]       1DelRd : 1MXRd[TRUE]
```

which will postpone the read operation if the value of 1MXRd is TRUE (see 2.4), that is in case of conflict with an ongoing write operation, and

```
[3405]       1DelWr : 1MXWr[TRUE] orelse 1MXsim[TRUE]
```

which will postpone the write operation in the corresponding case, and also when simultaneous requests are in conflict, in order to give priority to the read operation.

Authorisation is immediate if no delay is necessary; otherwise, it is given at the end of the conflicting operation, when EndWr or EndRd occurs:

```
[3406]       AutRd : any MReqRd whenno 1DelRd
                     orelse 1DelRd then EndWr
[3407]       AutWr : any MReqWr whenno 1DelWr
                     orelse 1DelWr then EndRd
```

The complete specification is then simply made up of definitions [3401] to [3407], ***in any order***, separated by "|" .

3.5 -The GO signal; recursive definitions. An example: the "totalizator"

A machine or a program are not always running. The symbol GO represents a fictitious signal occurring each time the machine is started or the program commences execution, before any signal is perceived. In this way, it will be possible to isolate the first dispatch of a signal after startup:

```
                     GO then A
```

or, taking into account an inhibitor H:

```
                     GO then A heed H
```

which is the first dispatch of signal A after startup without any intervening dispatch of H.

Internal contradictions are of course not acceptable in a specification, especially in absurd definitions such as the following:

```
                     A : (A orelse B) whenno C
```

which would imply: on a dispatch of B, A may occur when it doesn't; on a dispatch of C, A does not occur when it does. The name of the defined message may however appear in **then** or **next** expressions, in positions other than terminal, so that its dispatches take part in the selection of subsequent dispatches. This allows for carrying forward indefinitely step by step the effect of a dispatch. Thus for example, the definition

```
                     S2 : (S whenno S2) then S
```

perpetuates the memory of the first dispatch of S by eliminating every alternate one while preserving those of even rank. Because it applies over an unlimited number of dispatches, there exists no non-recursive definition of such a signal. Note that S2 may not be defined elsewhere; although it appears in the right-hand expression, its dispatches never occur except when selected amongst those of S. They are unambiguously the even ranking dispatches of S.

The signal defined by

```
                     C : (A orelse C) then B
```

is made up of the dispatches of B which are in immediate succession either to a dispatch of A or of C itself, that is which are subsequent to the first dispatch of A.

As an example of the use of GO and of recursivity with typed messages, the following definition of a message Accum is given, adding up the values carried by dispatches of two numerical messages Num1 and Num2 coming from two independent sources; the first dispatch of Accum occurs at startup, carrying the value 0, the following dispatches are those of the successive totals. Discounting the case of simultaneous dispatches of Num1 and Num2 for the time being (see [2407]):

```
                    Accum : 0 when GO
                              orelse Accum + next Num1
                              orelse Accum + next Num2
```

or, defining an intermediate message 1N1oN2:

```
                    Accum : 0 when GO orelse Accum + next 1N1oN2 |
                    1N1oN2 : Num1 orelse Num2
```

1N1oN2 is a numerical message as are N1 and N2. In this new form, it is now quite simple to take into account the case of simultaneous dispatches of Num1 and Num2:

```
                    Accum : 0 when GO orelse Accum + next Sum |
                    Sum : (Num1 + Num2) orelse Num1 orelse Num2
```

According to [2512], the sum Num1 + Num2 is preferred to either of its terms.

Internally to the specification, functions may be declared to yield a message, not simply a value as for banal functions (see 2.6); they define the result of calls by a message expression, using auxiliary definitions if necessary; all selectors are allowed. For example, with the definition

```
                    f_sum(a,b) : (a + b) orelse a orelse b |
```

the right part of the above definition of Sum becomes

```
                    f_sum(Num1,Num2)
```

which occurs each time Num1 and/or Num2 occur.

3.6 -In short, and now?

A specification is based on the image of the different cases to be taken into consideration, as formed in the mind of its author.

The number of cases to be considered is minimised as far as possible by the use of the current stroke as a temporal reference; nevertheless, their straightforward enumeration leads in general to an excessively long definition text, so that the author is inevitably led to regrouping them in some way or another. Some skill must inevitably be exercised in choosing intermediaries, but the author of a specification will never have any need for notions used only by designers of computing machines or programs.

We will see below that the basic notions adapted to the task are indeed readily converted or decomposed into notions appropriate to what we have called models, which are the forerunners of the final products.

Further steps towards producing programs are described in chapter 5. In section 5.7, the incorporation of sections written in procedural languages is envisaged.

Chapter 6 will present in complement a more sophisticated means of structuring behaviour specifications.

Chapter 4: Uncommitted models

As opposed to its specification, the ***model*** of a product takes into account principles common to digital machines as we effectively know how to construct them, but it otherwise remains uncommitted with respect to any actual design. It defines a mechanism satisfying the requirements defined by the specification; it may also be thought of as describing a machine made up of abstract components.

4.1 -State machines

At the level of the model, no distinction will be made between machines which are programmed or not; they will all be considered to be ***state machines***, whose ***global state*** may only change on receipt of dispatches of input signals. In certain states, a dispatch perceived may be transmitted instantaneously to one or more output port(s).

The global state can itself be decomposed into partial states each of which will correspond to a value, alphanumerical, numerical, Boolean or of any other data type, either simple or structured. Any change in the global state thus corresponds to a simultaneous change in one or more of the partial states. In an ideal state machine, transitions between states would be instantaneous, and simultaneous with the perception of the dispatches of the signals which provoked them; dispatches of output signals would also be emitted with zero delay. The model will thus take into account neither the inevitable delays which occur in real machines, nor the state transition duration in circuits, nor time-scattering in global state transitions implemented in von Neumann machines. The product itself will then later be designed in such a way as to guarantee to ***reproduce under all circumstances the sequence of global states as defined by the model.***

Some inputs referred to as ***static*** take on states defined by values of various types forced from outside, and intervene in defining the internal state transitions which may occur when input signals occur. States taken on by static outputs correspond to values depending on internal states.

The state machine will basically be described as a set of data memories of specific types, each one of which is able to hold as a partial state a value of its data type, together with a set of rules defining which dispatches are emitted in any configuration of partial states to be considered when input signals occur, and also which changes to the partial states will take place. In order to achieve this, a set of ***operations*** will be defined together with their symbolic representations, the ***operators***. As opposed to the “selections” and “selectors” previously discussed which are used only to describe behaviours, these entities will be used to describe abstract machines, or, and this amounts to the same thing, to set out the operation of target products independently of their detailed design.

4.2 -Time in a machine and the basic operations

Time in a real information processing machine is divided up into periods of equal or unequal duration during which the global state (that is, all the partial states which make it up) is defined and does not vary, and of intervals during which the global state is either undefined or varies. In our abstract machines, these intervals are not taken into account, nor are the durations of the periods which therefore may be considered as ***instants*** in time. Time thus becomes a sequence of instants.

Dispatches of incoming signals are generally changes of binary state. Temporal discrimination devices are used to guarantee that dispatches of the internal signals derived from these transitions coincide unambiguously with some machine instant. These devices are not part of the abstract machine described by a model. Similar devices ensure that static inputs always have an unambiguous value within their type at any given machine instant.

The usual operators (arithmetic, Boolean, etc.) deliver instantaneously the value of the product of those of the operands. They will be used in ***state expressions*** denoting instantaneous combinations of states. For example:

```
Ns : N1 + N2
```

will at each instant be the sum of the values of the numerical states `N1` and `N2`. This does not mean that the addition will be systematically repeated in the final product.

Likewise, the pair **`if…else`** chooses between two states of the same type depending on the value of a Boolean state:

```
E1 if B else E2
```

delivers the value of `E1` at the instants where `B` is `TRUE`, and that of `E2` at all other instants. This notation is extended without brackets:

```
[4201]      ... else Em if Bj else En  ==  ... else (Em if Bj else En)
```

The last part **`else`** may be omitted: the value of

```
E if B
```

is undefined whenever `B` is `FALSE`.

A single operator is sufficient to produce any durable state; this is achieved from a state of any type together with a ***sampling signal*** or ***sampler***:

```
E after S
```

where `E` is ***sampled*** by `S`, delivers a state which reproduces ***after*** (that is: at the instant following) ***each dispatch of S*** the value of `E` at the instant of this dispatch, and conserves it up to and including the next dispatch of `S`. This operation implies the existence of a data memory able to hold the value of a state of the type of `E`. This memory may be given a name by the following form of definition:

```
Em : E after S
```

If its type is **`Boolean`**, it holds at startup the value `FALSE`, else its contents are undefined. The operator **`after`** will be stronger than any ***op***, which is itself stronger than **`if`** and **`else`**. In a model expression, the hierarchy is thus (highest strength first):

`not`, arithmetic ***mop's***
`after`
`and`, **`or`**, arithmetic and relational ***op***'s as usual
`if`, **`else`**

Separation between successive states is strict:

```
N : (N + 1) after S
```

is a numerical state whose value will have increased by `1` after each dispatch of `S`. No operation can combine directly two successive states.

No double definition of a given state is allowed, so that the initial value of `N` above cannot be defined separately: practical recursive definitions involving **`after`** will be more complex.

Constant values may replace states in expressions.

4.3 -Internal signals

The internal signal `GO` (see 3.5) will be the state which is `TRUE` at startup, then `FALSE` from the next instant onwards.

At the output of the discrimination devices and then everywhere internal to the machine, ***a signal is a Boolean state with the value*** `TRUE` ***at the instants of its dispatches, and the value*** `FALSE` ***at all other instants***. This gives a meaning to the product

```
S1 after S2
```

It is a new Boolean state which is `FALSE` at startup, then takes on the value `TRUE` at the instant following a dispatch of the sampling signal `S2` if there was a simultaneous dispatch of `S1`, or the value `FALSE` otherwise, and conserving its value for the period ending just after the next dispatch of `S2`, if any.

Boolean operations are applicable between an internal signal and a Boolean state, and between internal signals. Instantaneous selectors as defined in 2.3 may thus be replaced in the model by (combinations of) Boolean operators; the symbol `->` will be used to indicate the transformation of a specification expression into a ***state expression*** within the model:

```
[4301]      S1 orelse S2  ->  S1 or S2
[4302]      S1 when S2  ->  S1 and S2
[4303]      S1 whenno S2  ->  S1 and not S2
```

An instantaneous expression is thus transformed by replacing each selector by the corresponding Boolean operator or combination of operators.

The basic operations are sufficient to recognise a first order succession and to define the corresponding signal of which a dispatch is emitted each time. This is achieved by a combination of an **after** operation saving the dispatch of the initial signal, with an **and** operation delivering a dispatch when the terminal signal occurs and the value saved was TRUE. A dispatch of the result must occur each time no mark (see 2.2) separates a dispatch of the terminal signal from a preceding dispatch of the initial signal, i.e. each time the immediate succession is completed. If A, B, and K are the initial, terminal and marker signals, the state expression

A **after** K **and** B

defines the required signal. The factor A **after** K takes on the state TRUE after each dispatch of A (which is always accompanied by K), and the state FALSE after each mark without A. A dispatch of the product is thus emitted if and only if B occurs in coincidence with the mark following A.

In order for the above to hold before the first dispatch of K, the data memory allocated to the operator **after** must initially contain the value FALSE: ***Boolean data memories will be systematically initialised to the value*** FALSE before startup.

The following illustrates the products of these operations; the value TRUE is indicated by the character "-":

```
instants            ................................................
GO                  -
A                         -   --    -      -   -  -    -   -     -
K                      -  - - --   --  - - -  --- - - --   -     -
A after K                  --  ----- ---    --- -  --   -----------
B                      -    -      -   - -     --     -    -     -
A after K and B             -      -   -        -          -     -
```

This establishes the transformation rule

[4304] A **then** B **among** K -> A **after** K **and** B

Transformation of the expression

A **then** B **heed** H

where H is an inhibitor signal, and whose marker is

any {A,B,H}

will then be carried out according to the rule

[4305] A **then** B **heed** H -> A **after** (A **or** B **or** H) **and** B

where the brackets enclose a model expression of the marker.

For **then** selections beyond first order (see 2.2), consider

S_0... **then** S_n **heed** H

When recognition of the succession S_0...,S_{n-1} on the referential {S_0...,S_n,H} has occurred, the completion of the overall succession takes place when a dispatch of S_n occurs immediately, with the referential unchanged. This is formalised in the equivalence

S_0... **then** S_n **heed** H == $1xs_{n-1}$ **then** S_n **among** {S_0...,S_n,H}

which is valid in association with the definition of the intermediate signal $1xs_{n-1}$. This decomposition has to be repeatedly applied to the expressions for the successive intermediates as defined by

$1xs_i$ **:** $1xs_{i-1}$ **then** S_i **among** {S_0...,S_n,H}

for i=n-1 to 1, ending with:

$1xs_1$ **:** S_0 **then** S_1 **among** {S_0...,S_n,H}

All right parts are first order **then** expressions, which will be readily transformed into model expressions. Left parts, unchanged, become names of intermediate Boolean states. Defining for conciseness the common sampler 2mk (names in the model not previously used in the specification will have as initial character "2"), the result is:

[4306] S_0... **then** S_n **heed** H -> $1xs_{n-1}$ **after** 2mk **and** S_n

The associated auxiliary definitions are:

```
1xs_i : 1xs_i-1 after 2mk and S_i |
................ |
1xs_1 : S_0 after 2mk and S_1 |
2mk : S_0... or S_n or H
```

4.4 -Typed internal messages

In the model, a typed message `Y` is split into a pair of states: its ***token*** `Y't`, which is an internal signal (a Boolean state), and its ***contents*** `Y'c`, which constitute a state of the same type as the message itself:

```
[4401]      Y  ->  {Y't; Y'c}
```

"`Y't`" and "`Y'c`" are just short notations for "`Y`'s token" and "`Y`'s contents" respectively.

Various dispatches correspond to different values of the contents. The token `Y't` is `TRUE` at each dispatch. At these instants, `Y'c` has the value carried by the message; ***at all other instants, its value is indifferent.***

Dispatches of the constituent `Y[v]` (see 2.4) are those of `Y't` for which `Y'c` has the value `v`; this is formalised in the transformation rule:

```
[4402]      Y[v]  ->  Y't and (Y'c = v)
```

A computation is thus required to obtain them. On the other hand, any operation defined on the type readily applies to contents; considering `[2602]`, an **`and`** operation between the tokens is then always required to complete the rule:

```
[4403]      Y1 op Y2  ->  {Y1't and Y2't; Y1'c op Y2'c}
```

This is an example of the transformation of a message expression from the specification into a pair of state expressions making up the corresponding expression in the model. As defined in 2.6, the contents have for value `v1` ***`op`*** `v2` when the selectands occur with values `v1` and `v2`.

A monadic operation ***`mop`*** applies to the contents alone:

```
[4404]      mop Y  ->  {Y't; mop Y'c}
```

When applied to messages, a function defined on the values of their types operates on contents; the **`and`** operator combines the tokens. For example:

```
[4405]      f(Y1,Y2,Y3)  ->  {Y1't and Y2't and Y3't; f(Y1'c,Y2'c,Y3'c)}
```

Expressions containing selectors must also be transformed into pairs of state expressions. In the case of the **`when`** selection applied to a typed message `Y1`, the token of the result follows from `[2505]`; each dispatch selected conserves its contents:

```
[4406]      Y1 when Y2  ->  {Y1't and Y2't; Y1'c}
```

For the **`whenno`** selection, the token follows from `[2506]`:

```
[4407]      Y1 whenno Y2  ->  {Y1't and not Y2't; Y1'c}
```

These two rules are extended to apply to chained **`when`** or **`whenno`** selections. The contents of the result are always those of the first selectand.

Whatever the type common to `Y1` and `Y2`, preference applies as follows:

```
[4408]      Y1 orelse Y2  ->  {Y1't or Y2't; Y1'c if Y1't else Y2'c}
```

where the token follows from `[2511]`. It can be seen that the model expression of the constituent of index `v` according to `[4402]`,

```
            (Y1't or Y2't) and (v = (Y1'c if Y1't else Y2'c))
```

gives the same value as that derived from transforming `[2603]`:

```
            Y1't and (v = Y1'c) or Y2't and (v = Y2'c) and not Y1't
```

Rule `[4408]` extends to three terms as follows:

```
            Y1 orelse (Y2 orelse Y3)
  -> {Y1't or (Y2't or Y3't); Y1'c if Y1't else (Y2'c if Y2't else Y3'c)}
```

or, according to [4201], without brackets:

```
[4409]     Y1 orelse Y2 orelse Y3
-> {Y1't or Y2't or Y3't; Y1'c if Y1't else Y2'c if Y2't else Y3'c}
```

The signal expression

any Y_1 **then any** Y_2 **heed** H

given by [2606] for the token of

Y_1 ***op*** **next** Y_2 **heed** H

will be transformed according to [4305]. Moreover, in order to implement the ***op*** **next** selection in the model, the value carried by a dispatch of Y1 must be retained at least until the next mark so that its contents may be combined with the contents of a later dispatch; its token or the marker may be used as a sampling signal to achieve this retention, since they occur at each dispatch and never between marks. Choosing to define the marker

2K **:** Y_1't **or** Y_2't **or** H't

the rule corresponding to [4305] is:

[4410] Y_1 ***op*** **next** Y_2 **heed** H
-> {Y_1't **after** 2K **and** Y_2't; Y_1'c **after** 2K ***op*** Y_2'c} |

Similar rules with different contents in the right-hand part will be used in the case of **then** and **when next** selectors occurring instead of ***op*** **next**:

Y_1 **then** Y_2 **heed** H
-> {Y_1't **after** 2K **and** Y_2't; Y_2'c}

and

Y_1 **when next** Y_2 **heed** H
-> {Y_1't **after** 2K **and** Y_2't; Y_1'c **after** 2K}

Rule [4410] may be extended in a similar way to [4305]:

[4411] Y_0**...** ***op*** **next** Y_n **heed** H
-> {$1xs_{n-1}$ **after** 2mk **and** Y_n't; $1xs_{n-1}$'c **after** 2mk ***op***$_n$ Y_n'c}

The set of auxiliary definitions is now:

$1xs_i$ **:** {$1xs_{i-1}$ **after** 2mk **and** Y_i't; $1xs_{i-1}$'c **after** 2mk ***op***$_i$ Y_i'c} |
.... |
$1xs_1$ **:** (Y_0't **after** 2mk **and** Y_1't; Y_0'c **after** 2mk ***op***$_1$ Y_1'c} |
2mk **:** Y_0't**...** **or** Y_n't **or** H't

Expressions will be somewhat simpler in the case of a **then** or **when next** selector occurring instead of ***op***$_i$ **next**, respectively:

$1xs_i$ **:** (Y_{i-1}'t **after** 2mk **and** Y_i't; Y_i'c}

and

$1xs_i$ **:** (Y_{i-1}'t **after** 2mk **and** Y_i't; Y_{i-1}'c **after** 2mk}

Rules [4410] and [4411] hold for ***op*** **next** selections expressed in the full form.

4.5 -Transforming complex message expressions

Rules [4401] to [4411] transform any expression of a simple selection of typed messages into a ***model expression***, which consists in a pair of state expressions between braces: "{...}". The symbols

Y, Y1...Y3, Ya...Yn

may on the one hand be message names such as

msgname

In this case, the convention in use in this document assigns the name of the message to the token, and the same name suffixed _c to the contents, if any:

msgname -> {msgname; msgname_c}

Symbols Y...Yn may also stand for basic expressions such as

{name; vxpr}

where name is the token's name, and vxpr a value expression.

Symbols `Y...Yn` may more generally stand for transformed message expressions valid in the model, in the general form:

```
                {xT; xC}
```

where `xT` is a Boolean state expression and `xC` a state expression of any type. In such case, "`'t`" and "`'c`" pick out the token expression and, if any, the content expression, following the additional rules:

```
    [4501]      {xT; xC}'t  ->  xT
    [4502]      {xT; xC}'c  ->  xC
```

Finally, symbols `Y1...Yn` in transformation rules may stand for untransformed expressions such as:

```
                (Y1 op Y2)
```

These expressions will be transformed by "`'t`" and "`'c`" according to the appropriate rules in 4.4, which may be rewritten for that purpose. For example, `[4403]` will be split into:

```
    [4503]      (Y1 op Y2)'t  ->  Y1't and Y2't
    [4504]      (Y1 op Y2)'c  ->  Y1'c op Y2'c
```

A complex specification expression is a notation for the result of multiple selections, each applied to selectands which may themselves be either message names, basic expressions or expressions for unnamed intermediary results. Its transformation into a model expression will be carried out in several steps.

To avoid excessive multiplication of transformation rules, any specification signal involved in mixed selections will be considered as an empty message, following the preliminary transform:

```
    [4505]      S  ->  {S; }
```

The transformation of an **any** selection will produce such a message:

```
    [4506]      any {Y1... ,Yn}  ->  {Y1't... or Yn't; }
```

Used as a selectand in a message expression, a value expression *vxpr* will first be transformed according to the rule:

```
    [4507]      vxpr  ->  {STROKE; vxpr}
```

4.6 -Models of a keypad entry system and of a "totalizator"

Consider the specification of the keypad entry system given by `[3101]`:

```
           OPEN : C4 then C7 then C9 then C0 heed {  C1,C2,C3  ,C5,C6  ,C8  }
```

Applied to the right hand expression, rule `[4306]` leads to the model definition of the Boolean state `OPEN`, associated to three auxiliary definitions:

```
           OPEN : 1xs_479 after 2mk and C0 |
           1xs_479 : 1xs_47 after 2mk and C9 |
           1xs_47 : C4 after 2mk and C7 |
           2mk : C4 or C7 or C9 or C0 or C1 or C2 or C3 or C5 or C6 or C8
```

Each intermediate is a Boolean state. The order of the definitions is indifferent.

Consider now the specification from 3.5 of the "totalizator"

```
                Accum :0 when GO orelse Accum + next Sum |
                Sum : (Num1 + Num2) orelse Num1 orelse Num2
```

where `Num1` and `Num2` are the input messages, and `Accum` the output message.

Transformation will consist first of applying rule `[4411]` to the subexpression `Accum + next Sum`, which gives the model expression

```
                {Accum after (Accum or Sum) and Sum;
                 Accum_c after Accum + Sum_c}
```

Rule `[4403]` applied to the subexpression `Num1 + Num2` gives the model expression

```
                {Num1 and Num2; Num1_c + Num2_c}
```

Finally, the extended rule [4408], applied to the resulting expressions for Sum and Accum, gives the following set of definitions after separating tokens and contents:

```
Accum : GO or Accum after (Accum or Sum) and Sum |
Accum_c : 0 if GO else Accum_c after Accum + Sum_c |
Sum : Num1 and Num2 or Num1 or Num2 |
Sum_c : Num1_c + Num2_c if Num1 and Num2
            else Num1_c if Num1
            else Num2_c
```

If there had been calls of message functions (see 3.5) in the specification, they would have been developed before transformation, according to the declarations of the functions.

The separation of tokens and contents corresponds to the representation of typed messages, internally and at the frontier of the abstract machine. Tokens are Boolean states; in the above example, all contents are numerical states.

It sometimes occurs that the model language is the most convenient to write some parts of a specification. Its use presents no difficulty, since model operators cannot be confused with selectors.

4.7 -Semantic simplification

Models may be improved by applying rules specific to the operations implied.

As far as Booleans are concerned, such improvements would be trivial except for **after** operations, for which equivalences must be established beyond Boolean algebra. Two state expressions will be said to be equivalent (symbol ==) if their values are equal at any instant.

In the first place, if B is a Boolean state, the definition of the **after** operation implies

```
[4700]     B after S == (B and S) after S
```

since the value of B is indifferent unless the sampler S is true. The following equivalences also hold:

```
[4701]     B1 after S and B2 after S == (B1 and B2) after S
[4702]     B1 after S or B2 after S == (B1 or B2) after S
[4703]     not(B after S)  ==  GO or not B after S
```

An equivalence similar to [4703] holds for Boolean function calls. Each equivalence derives at the first instant from the initialisation to FALSE of all Boolean memories (and therefore of all Boolean products of the **after** operations); from then on, operations on the values conserved reproduce the results conserved, so that the equivalence continues to hold.

Keeping in mind that non-Boolean values stored in **after** operators are undefined at startup, equivalences analogous to [4700] to [4703] apply to states E, E1, E2 of any type:

```
[4705]     E after S == (E if S) after S
[4706]     E1 after S op E2 after S == (E1 op E2) after S
[4707]     mop(E after S)  ==  mop E after S
```

Considering **after** operation on signals, the product

```
           S after S
```

is the state which is FALSE from startup, then TRUE at any instant following the first dispatch of S, as a result of the contiguity of the TRUE periods initiated by consecutive dispatches of S:

```
[4708]     S after S  ==  TRUE after S
```

In particular,

```
[4709]     GO after GO  ==  TRUE after GO
```

is the state which is FALSE at startup, then TRUE at other instants.

All signals perceived occur after GO, so that , if P is one of them or is simultaneous to one of them:

```
[4710]      GO and P  ==  FALSE
[4711]      GO and not P  ==  GO
[4712]      P and not GO  ==  P
[4713]      GO after P  ==  FALSE
[4714]      P after GO  ==  FALSE
[4715]      P and GO after GO  ==  P
[4716]      P and not(GO after GO)  ==  FALSE
```

Considering `[4708]` and `[4709]`,

```
[4717]      GO after GO or P after P  ==  GO after GO  ==  TRUE after GO
[4718]      GO after GO and P after P  ==  P after P  ==  TRUE after P
```

Another point is that a dispatch of a sampler is ineffective unless the value of the sampled state is different from the memorised value. Thus, using the symbol == again for equivalent definitions, and the Pascal operator <> for "unequal to":

```
[4719]      Em : E after S  ==  Em : E after (S and (E <> Em))
[4720]      Em : E after S  ==  Em : (E if S and (E <> Em)) after S
```

which hold for any type. The case of the recursive definition of a Boolean state

```
            Bm : (B or Bm) after S
```

is of special interest. Since

```
            (B or Bm) <> Bm  ==  B and not Bm
```

only the dispatches of `S` occuring when `B` is `TRUE` can be effective, so that

```
            Bm : (B or Bm) after S  ==  Bm : (B or Bm) after (S and B)
```

Hence, since `(B or Bm)` is `TRUE` whenever `(S and B)` occurs:

```
[4721]      Bm : (B or Bm) after S  ==  Bm : TRUE after (S and B)
```

Otherwise, concerning the model expression of a typed message, the equivalence

```
[4722]      {T; C}  ==  {T; C if T}
```

formalises the fact that the value of its contents matters only at instants at which the token is dispatched.

As an example of semantic simplification, consider the model in 4.6 derived from the specification of the totalizator.

1) Using as auxiliary definition

```
            BB : Accum after (Accum or Sum)
```

the definition of the token becomes

```
            Accum : GO or BB and Sum
```

so that, considering Boolean rules `[4710]` and `[4700]`,

```
            BB : (GO or BB) after (GO or Sum)
```

or, using `[4721]`, then `[4710]` and again Boolean rules,

```
            BB : TRUE after GO
```

and, considering once more`[4710]`:

```
            Accum : GO or Sum
```

2) Considering Boolean rules, the term `Num1 and Num2` is unnecessary in the expression for `Sum`.

The simplified model will then be:

```
            Accum : GO or Sum |
            Accum_c : 0 if GO else Accum_c after Accum + Sum_c |
            Sum : Num1 or Num2 |
            Sum_c : Num1_c + Num2_c if Num1 and Num2
                        else Num1_c if Num1
                        else Num2_c
```

Chapter 5: Programs

We saw in chapter 4 that the specification of a machine's behaviour in terms of selections of messages may be transformed automatically into a model which is the description of an abstract machine in terms of commonplace operations together with the memorisation of states. In this chapter, we will now show how the program for a conventional computer can be derived directly from this model.

5.1 -Alternate phases

The states from the model subsist in the actual computer. The necessity thus remains for strict separation between successive memorised states: this is achieved by distinguishing between two classes of memory, the contents of which change during alternate dedicated phases of the machine's operation.

On the one hand the ***liaison memories*** store the products of the instantaneous operations as they are evaluated so that they become available as inputs to further evaluations. Amongst these, the ***terminal memories*** receive the ***terminal results***, those which are then picked up in the model by the **`after`** operators.

On the other hand the ***principal memories*** are allocated to the **`after`** operators themselves, in order to conserve the states which must be memorised.

In the first of the alternating phases of operation, states previously saved in the principal memories are used as inputs to compute results which are placed in the liaison memories as they become available: this is the compute phase, or ***"tick" phase***, the results of which will be recopied.

Before another compute phase can take place, the principal memories must be updated in the memorisation phase, or ***"tock" phase***, which sets up the inputs for the next computation. Memorisation consists in copying the contents of each terminal memory into the principal memory allocated to the corresponding **`after`** operator, each time the sampler, which is itself a terminal result, occurs (that is, has the value `TRUE` at the end of "tick").

As can be seen, the processor, or the set of available processors, is used alternatively for computation and for memorisation. Phases of one of the two types can trigger directly phases of the other type, ***without any clock signals***. The only constraints are that the state of the principal memories must not change during the compute phase, nor must terminal results change during memorisation. This sequence of alternate types of phases corresponds to a cyclical pattern of activity. In a model, the dispatch of a signal is `TRUE` at an instant. In the computer, this will correspond to a `TRUE` state for the operations of a single cycle. In this way, dispatches will effectively be "simultaneous" if both corresponding states are `TRUE` for the operations in which they are involved during a cycle

The program will be made up of two parts, corresponding to the two phases. Communication between them will be achieved by names given to the instantaneous results (all delivered in "tick" phases) if they were not named in the model and if they are implicated in memorisations (which all take place in "tock" phases), and also to the results of memorisations (all used in "tick" phases).

This applies to expressions immediately preceding or following **after**, and to the results of the **after** operations themselves. When a result saved in a principal memory is used in a "tock" phase without change, a simple copy which is also given a name is made during the "tick" phase.

5.2 -Adapting a model: the keypad entry system

Consider the model derived in 4.6:

```
OPEN : 1xs_479 after 2mk and C0 |
1xs_479 : 1xs_47 after 2mk and C9 |
1xs_47 : C4 after 2mk and C7 |
2mk : C4 or C7 or C9 or C0 or C1 or C2 or C3 or C5 or C6 or C8
```

Following the previous discussion, names are also required (prefix "a_") for the results memorised by **after**.The *adapted model* is thus:

```
OPEN : a_1xs_479 and C0 |
1xs_479 : a_1xs_47 and C9 |
1xs_47 : a_C4 and C7 |
2mk : C4 or C7 or C9 or C0 or C1 or C2 or C3 or C5 or C6 or C8 |
a_1xs_479 : 1xs_479 after 2mk |
a_1xs_47 : 1xs_47 after 2mk |
a_C4 : C4 after 2mk
```

In the computer, names of the form `Ci` are memories allocated to external signals from the keypad; names prefixed "a_" are principal memories allocated to the products of **after** operators. `OPEN`, `2mk` and names of the form `1xs_i` are terminal liaison memories.

5.3 -Eliminating `after` operators; computing at first request

Recurrence of the phases is achieved by an endless loop of imperative instructions including a "tick" part and a "tock" part. Transposing a definition such as

```
A : E after R
```

gives the instruction

```
if R then A := E
```

which, following the discussion in 5.1, must be executed in phase "tock". Conversely, `E` and `R` must be computed in "tick" phase. Overall, this gives:

```
        "tick" part:
(if there exists a definition of R)       compute R
(if there exists a definition of E)       if R then compute E
        "tock" part:                      if R then A := E
```

In order to avoid unnecessary computation, values to be memorised are computed only during cycles for which their sampling signals are `TRUE`. As mentioned, **compute** may just be a simple copy operation.

The interface devices are assumed to set all `Ci` to `TRUE` or to `FALSE` for each cycle depending upon whether the corresponding input signals from the keypad occur or not, and to emit a dispatch of `OPEN` for each cycle in which its memory has been set to `TRUE`.

The principal memories are implicitly initialised to `FALSE` as they are of type Boolean; the result memories require no initialisation as their contents are never used before being computed.

Retaining those definitions which do not contain the **after** operator, a partially procedural program is obtained, which includes definitions with instantaneous expressions as their right-hand sides, and an imperative loop ***which computes in phase "tick" the values to be memorised or emitted and carries out in phase "tock" the memorisations***:

```
OPEN : a_1xs_479 and C0 |
1xs_479 : a_1xs_47 and C9 |
1xs_47 : a_C4 and C7 |
2mk : C4 or C7 or C9 or C0 or C1 or C2 or C3 or C5 or C6 or C8
repeat
            {"tick" part:}
      compute OPEN,
      compute 2mk; if 2mk then compute 1xs_479,
      compute 2mk; if 2mk then compute 1xs_47
      ;   {"tock" part:}
      if 2mk then a_1xs_479 := 1xs_479,
      if 2mk then a_1xs_47 := 1xs_47,
      if 2mk then a_C4 := C4
endlessly
```

The **after** elimination process will have generated the two parts in parallel.

The definitions of `OPEN`, `2mk`, `1xs_47` and `1xs_479`, in which right-hand sides are instantaneous expressions, specify the computations to be carried out ***when their values are needed after* compute*, or in an expression being evaluated*** (this last case does not occur in the example under consideration). Computation is carried out ***at the first request*** in each cycle, and always in phase "tick". Results remain in liaison memories until the end of the cycle, where they are thus available at other requests in the "tick" phase and in the "tock" phase.

As a general rule, input data and results will be of one of the usual types or of a specific previously defined type; there may possibly be several sampling signal expressions. Memories for contents need not be initialised. If there are recursively defined messages, the instantaneous definitions derived from splitting these up are never recursive, neither directly nor indirectly.

5.4 -Instruction sequence

In the same way as for the order of definitions, the order of the lines in each of the two parts considered separately is of no importance.

However complex their definitions may be, dispatches of all internal signals always coincide with dispatches of external signals, which will only be effective in the first cycle following their occurrence. As a result, if no external message has occurred during a cycle, no state changes and no emissions would take place in the cycle to come, which thus need not be triggered. In such a case, the next cycle will be initiated by the next dispatch.

As an illustration, consider the Pascal-like wholly sequential program which can be derived from the keypad entry system program in 5.3 above. The "tick" and "tock" parts respectively carrying out computation and memorisation will again be found, together with additional instructions ensuring initialisation and communication:

```
        a_C4, a_1xs_47, a_1xs_479 := FALSE;
        C0, C1, C2, C3, C4, C5, C6, C7, C8, C9 := FALSE;
        repeat
              {"tick" part:}
          OPEN := a_1xs_479 and C0;
          2mk := C4 or C7 or C9 or C0 or C1 or C2 or C3 or C5 or C6 or C8;
          if 2mk then 1xs_479 := a_1xs_47 and C9;
          if 2mk then 1xs_47 := a_C4 and C7;
              {"tock" part:}
          dispatch OPEN;
          if 2mk then a_1xs_479 := 1xs_479;
          if 2mk then a_1xs_47 := 1xs_47;
          if 2mk then a_C4 := C4;
              {acquisition:}
          accept C0, C1, C2, C3, C4, C5, C6, C7, C8, C9
        endlessly
```

The sequentialisation process will have first eliminated one of the two

```
                compute 2mk
```

instructions from the original “tick” part.

The expressions for OPEN, 2mk and for the 1xs_i states are now included in the “tick” part. Since 2mk appears in the **if** instructions, the order of the definitions could not be preserved in the “tick” part. As a general rule permutations are necessary but, since there are no cyclic references in instantaneous definitions, there always exists a satisfactory order.

The sequence of instructions in the “tock” part remains unconstrained. The **dispatch** instruction emits a dispatch of the OPEN signal each time the OPEN memory is found holding the TRUE state. In this example, no signal may ever occur at first execution of the main loop.

The **accept** instruction sets the value TRUE for memories allocated to input signals which have occurred since its previous execution, and the value FALSE for all others. These memorised states then remain unchanged for a whole cycle. An individual buffer memory initialised to FALSE at startup may be assigned to each input signal, which is then set to TRUE at each corresponding dispatch and reset by **accept**. If however the order of input must be preserved as accurately as possible, a single common input queue will be used. Perception of a dispatch, or of several simultaneous dispatches, will then add a word of ten Boolean values to the queue, with the value TRUE for any signal or signals which have occurred, FALSE for the others.

5.5 -Another model adapted: a program for the “totalizator”

Consider again the model from 4.7. The expansion of **after** operations requires the use of names (prefixed “a_”) for the corresponding products in the adapted model:

```
                Accum : GO or Sum |
                Accum_c : 0 if GO else a_Accum_c + Sum_c |
                Sum : Num1 or Num2 |
                Sum_c : Num1_c + Num2_c if Num1 and Num2
                              else Num1_c if Num1
                              else Num2_c |
                a_Accum_c : Accum_c after Accum
```

Either or both contents Num1_c and Num2_c are expected to have been updated and either or both tokens Num1 and Num2 set to TRUE for a whole cycle whenever either or both messages Num1 and Num2 have occurred; a dispatch of Accum carrying the value of Accum_c is expected to be emitted at each cycle for which Accum is TRUE. The memorised Boolean state a_Accum_c may have any value at startup. GO is TRUE at the first cycle.

In the same way as for the keypad entry system, a semi-procedural program can then be derived from this adapted model according to the rule in 5.3. It will be:

```
            Accum : GO or Sum |
            Accum_c : 0 if GO else a_Accum_c + Sum_c |
            Sum : Num1 or Num2 |
            Sum_c : Num1_c + Num2_c if Num1 and Num2
                        else Num1_c if Num1
                        else Num2_c
            repeat
                    {"tick" part:}
                compute Accum; if Accum then compute Accum_c
                ;   {"tock" part:}
                if Accum then a_Accum_c := Accum_c
            endlessly
```

An extra memory is then allocated to the GO signal, which occurs once at each startup (see 3.5). A purely sequential program may then be obtained by providing for initialisation, by adding **accept** and **dispatch** instructions, and finally by integrating the definitions into the instructions of the "tick" part:

```
                   {initialisation:}
            GO := TRUE;
            Num1, Num2 := FALSE;
            repeat
                    {"tick" part:}
                Sum := Num1 or Num2;
                if Sum then
                    Sum_c := Num1_c + Num2_c if Num1 and Num2
                                else Num1_c if Num1
                                else Num2_c;
                Accum := GO or Sum;
                if Accum then Accum_c := 0 if GO else a_Accum_c + Sum_c;
                    {"tock" part:}
                if Accum then dispatch Accum;
                if Accum then a_Accum_c := Accum_c;
                    {acquisition:}
                GO := FALSE;
                accept Num1, Num2
            endlessly
```

For conditional expressions, the notation:

```
                X1 if B else X2
```

has been preferred to those of the programming languages Algol and C:

```
                if B then X1 else X2
```

and

```
                B ? X1 : X2
```

Banal function calls, if any, would remain unchanged in the "tick" part, in appropriate positions.

The principal Boolean memories only are initialised; they are set to FALSE, except for GO which is set to TRUE.

Instructions in the "tick" part are not in the same order as the corresponding definitions, since Sum and Sum_c must be ready in time for the computation of Accum and Accum_c. As has already been pointed out, there always exists a satisfactory order.

The "tock" part includes an instruction setting the GO memory to FALSE for all cycles other than the first. Each time Accum is found with the value TRUE, the **dispatch** instruction emits a dispatch of Accum. Besides, a_Accum and a_Accum_c are updated only in such a case; this will hold for any message.

When **accept** does not find either Num1 or Num2 in the input buffer, Num1 or Num2 respectively will be set to FALSE, but the corresponding contents Num1_c or Num2_c will remain unchanged.

5.6 -Peripherals, files and merged activities

Up to now, we have considered that input messages occur spontaneously, whereas in actual fact common peripheral devices operate on request from a program: a traditional **read** instruction will expand into **dispatch** and **accept** instructions. A single ongoing activity will thus be able to handle several peripherals.

Certain sources such as those submitting the read and write requests to the "mutual exclusion" server in 3.4 may be virtual machines with separate specifications. They will then also correspond to cyclical activities, and their programs, just after elimination of the **after** operations, will each be made up of a set of definitions, followed by a "tick" part and a "tock" part in an endless loop. The effect of such a program depends only on the alternate execution of these two parts in the loop, so that the instructions in each part can be merged with instructions in the corresponding part of other programs or activities without risk of interference: the activities themselves are thus merged.

Co-operation between such merged activities then clearly no longer requires emitting, receiving or buffering: data which previously needed to be transmitted becomes directly accessible. Finally, if the machine cannot provide adequate performance when operating in the compute on first request mode, overall sequentialisation can be carried out. External relations are achieved with a single **dispatch** instruction and a single **accept** instruction. No waiting occurs unless no dispatch was perceived during the cycle which is being completed.

Co-operating processes are no longer interlaced in an unstructured manner, so that critical sections need not be protected by semaphores.

5.7 -Use of dedicated functions; specification of a reservation program

If not available, banal functions (computing only content values) used in expressions must be defined externally to the specification in an appropriate language which may be procedural, such as C and Pascal. A function f may have been declared, in C,

```
f(x1,x2,x3,pz1,pz2)
```

where x1,x2,x3 are value parameters and pz1,pz2 pointer parameters, and called by

```
f(Y1,Y2,Y3,&V1,&V2)
```

where Y1,Y2,Y3 are message expressions and V1,V2 message names or constants, its result is first the signal

```
STROKE when any Y1 when any Y2 when any Y3
```

reduced to STROKE if there is no value parameter, and which is the token of a message f carrying the returned value, if any. It is also the token of V1,V2, if they are messages. The call may also have updated their contents, initially undefined, by assignment to parameters pz1,pz2: the call expression is then a definition of V1 and/or V2.

To the two forms of function declaration in Pascal,

```
function f(x1,x2,x3; var z1,z2)
procedure f(x1,x2,x3; var z1,z2)
```

the latter denoting that the body assigns no value to f, correspond calls of the same form:

```
f(Y1,Y2,Y3,V1,V2)
```

which will be preferred to the C form; as is the case for other banal calls, they remain unchanged in the "tick" part.

Used in specifications, such functions may only process stored data, the contents of arguments or data to which they point. ***They may not use system calls or traditional run-time library calls*** to get data from, or send data to, files or peripheral devices.

A few re-entrant functions will have been programmed at low level to perform I/O operations, with the requirement that ***answers to different calls of I/O functions never be dispatched at the same stroke. Contrary to ordinary functions, they do not dispatch their results at the stroke of the call***, but at a later instant, which becomes a stroke.

Let us design a crude specification for a reservation program (extendable to cancellation), operated from several consoles numbered `1...n`, and using dedicated functions to perform all computations required. Instead of the traditional `n` parallel tasks, there will be `n` activities in a single task

Consoles and files will be accessed through calls of the following re-entrant I/O functions:

```
write_string(do,file,string var done)
write_line(do,file,string var done)
read_line(do,file var string,done)
put_record(do,file,key,record var done)
get_record(do,file,key var record,done)
```

In the value part, `do` is the ***request message***. In the **`var`** part, `done` occurs in answer to it, carrying a copy of the contents of `do`; the other **`var`** argument, if any, occurs in simultaneity with it.

In the intended specification, there will be as many activities as consoles concerned. Its skeleton will be made up of I/O function calls; arguments not provided by other I/O calls will be explicitely defined, some of them by bare function calls. Altogether:

```
inibook(GO) |
prompt: 1 when GO
      orelse (start+1) when (start<n) orelse loop1 orelse loop2 |
clear(prompt) |
write_string(prompt,console(prompt),'date and flight? ',start) |
read_line(start,console(start),flight,flight_read) |
seat_nb : get_key(flight_read,flight) |
get_record(flight_read when (seat_nb<>NIL),
            flight,seat_nb,record,record_got) |
seat_free : record.name='' |
write_string(record_got when seat_free,console(record_got),
             'your name, please? ',name_request) |
read_line(name_request,console(name_request),name,name_read) |
book(name,record) |
put_record(name_read,kept(flight),kept(seat_nb),record,OK)|
write_line(OK,console(OK),
        'OK, your seat number is '+ kept(seat_nb),loop1) |
no_seat : flight_read when seat_nb=NIL
      orelse record_got whenno seat_free |
write_line(no_seat,console(no_seat),
            'sorry, flight booked up',loop2)
```

I/O results may be queued. In the definition

```
seat_nb : get_key(flight_read,flight)
```

the dedicated function `get_key` uses, as soon as `flight_read` occurs, an internal algorithm to choose a record in `flight`, and dispatches the `get_key` message with key or `NIL` as contents. This record is temporarily reserved to the use of the activity whose number is carried by `flight_read`. To that end, the function `get_key` is programmed not to deliver again the key to this record until it has been freed by

```
clear(prompt)
```
when `prompt` carries the same activity number. There exists, in a space global to all dedicated functions, a `2*n`-element array to store a flight identification and a seat number for every activity. The call of `clear` dispatches nothing, nor does
```
inibook(GO)
```
which initialises at startup other variables in the global space; nor does
```
book(name,record)
```
which assigns, when `name` occurs, its contents to the field `name` of `record`, and redispatches the latter. The function
```
console(ay)
```
used in I/O calls, identifies the console alloted to the activity `ay`. The calls

`kept(flight)` or `kept(seat_nb)`

deliver a value stored by `get_key` in the global space.

Chapter 6: Time-windows

In practice, it will be necessary to develop a specification language offering various convenient features with the aim of achieving greater clarity and modularity in specification texts. Some such features may be inspired by those from programming languages. This is however not the case of the extension considered, together with its influence on the models and programs produced, in this chapter.

6.1 -Prospecting expressions: a new keypad entry system

A referential enumerates the messages intervening in a selection, but does not distinguish between their different dispatches. ***Prospection*** in addition delimits periods of time (or ***time-windows***) over which dispatches of messages will be taken into account.

A time-window opens immediately after the dispatch of a ***trigger*** message such as `T`, and shuts immediately after the dispatch of a ***limiter*** message such as `L`:

```
T                   *         *   * *     *   *   *
L                          *           * *    *          *
1st  time-window     -------
2nd  time-window               ---------
3rd  time-window                           ----
4th  time-window                               -----------
```

The trigger `T` has no effect on an open window; between two windows, the limiter `L` has no effect. When `T` occurs in simultaneity with `L`, any ongoing window shuts and a new window opens immediately. This is shown above where four distinct time-windows are indicated, the last two being contiguous. ***Each one lasts from a dispatch (excluded) of the trigger up to the following dispatch (included) of the limiter.***

An expression (possibly reduced to the simple name of a message) may be the ***object*** of a prospection, whose time-windows are then said to ***cover*** certain dispatches of its ***components***, that is of the messages whose names appear in it. The selections carried out ***then apply only to the dispatches covered, and any dispatch of their results depends solely on dispatches of those components covered by the same time-window***. As will be seen in 6.3, a specific exception to this rule will be defined to allow the dispatch of the trigger which opened a time-window to be combined with dispatches covered by the latter.

The following notation is used to indicate a prospection triggered by `trigger`, applied to `object`, and limited by `limiter`:

```
                    trigger prospects (:object:) till limiter
```

is a ***prospecting expression*** or ***prospective***. The highest strength is given to **`prospects`** and **`till`**, so that `trigger` and `limiter` should be simple names or bracketed expressions, indexed or not, or **`any`** expressions, or function calls. The prospected expression will always be between the special compound brackets "`(:`" and "`:)`". A prospective may be used in the same way as any other expression, in particular as trigger, object or limiter. Unlike `trigger` and `limiter`, `object` does not denote a message by itself, but only by association with `trigger` and `limiter`.

Consider once again the keypad entry system. In the formal specification `[3101]` for the code `4790`, the expression for `OPEN` may be considered as being under permanent prospection through a time-window opened at startup, as all the dispatches of its components are taken into account. Changing this specification to reflect other codes is not however completely trivial, even though there is no change in the referential. For example, to specify code `2222`, the definition

```
[6101]      OPEN : C2 then C2 then C2 then C2
                              heed {C0,C1,C3,C4,C5,C6,C7,C8,C9}
```

signifies that the door will open each time the "2" key has been pressed four times at least in immediate succession. With this early version of the system as defined, and except at startup, visitors have the benefit of keystrokes preceding their own attempts. An authorised visitor will thus succeed the first time he or she presses the `'2'` key, if no other key has been pressed since the last entry; and an intruder will generally have one chance in ten of succeeding instead of the one chance in ten thousand the four digit code should guarantee.

To avoid this, the above right-hand side will be prospected through a window closed by a dispatch of `OPEN`, after which a new window will open; for any code `pqrs` the definition becomes

```
[6102]      OPEN : (GO orelse OPEN)
                       prospects (:Cp then Cq then Cr then Cs
                                    heed {C0,C1,C2,C3,C4,C5,C6,C7,C8,C9}:)
                       till OPEN
```

The `GO` symbol (see 3.5) represents startup of the machine or launch of the program. `OPEN` is used recursively to limit the prospection (without inhibiting itself, as the limiter does not close the window instantaneously), while also retriggering it. As with the early version of the system, the visitor may make any number of errors: the door opens as soon as he or she has pressed the four keys for the code in the correct order. Moreover, the door never opens before the fourth keystroke after the last successful attempt, whatever the chosen code may be and whatever may have happened previously.

6.2 -Nothing is put into question

Prospection is not a new primitive: any prospective can be replaced by an expression combining selections defined in chapter 2, applied to the trigger and limiter expressions and to the components of the object expression. The notation is vindicated by its concision, as will be clear from the following three examples of prospection on typed messages.

Consider first the definition

```
[6201]      YP : trg prospects (:Y:) till lim
```

where the prospected expression is reduced to a simple message name `Y`, and which signifies that `YP` is made up of the dispatches of `Y` covered by the time-window(s) opened by `trg` and shut by `lim`. The equivalent definition without prospection would be recursive:

```
[6202]      YP : trg then Y heed lim
                    orelse (YP whenno lim) then Y heed lim
```

It specifies how dispatches of `YP` may be selected among those of `Y`: starting with the first, and then from one to the next as long as `lim` has not occurred.

As a second example, consider the prospective

```
[6203]      trg prospects (:Y1 isel Y2:) till lim
```

where ***isel*** is any operator or instantaneous selector. Any particular dispatch of the result is only dependent on simultaneous dispatches of the selectands, which are covered by the same time-window as itself: there is no need to eliminate effects of dispatches occurring in previous windows, so that an equivalent expression is

```
[6204]      trg prospects (:Y1:) till lim isel trg prospects (:Y2:) till lim
```

This conclusion extends to instantaneous selections of expressions `x1`, `x2`:

```
[6205]      trg prospects (:x1 isel x2:) till lim
         == trg prospects (:x1:) till lim isel trg prospects (:x2:) till lim
```

Consider as the third example the expression

```
[6206]      trg prospects
              (:xa opb next xb ... opm next xm opn next xn heed H:)
            till lim
```

where ***op***b,... ***op***n are operators. Its dispatches occur whenever the dispatches of `xa`, `xb`,... and finally of `xn` have just occurred in immediate succession in the same time-window. Unlike the inhibitor `H`, the limiter prevents the dispatch of the result of the prospection each time it occurs between the trigger and the terminal selectand `xn`, including the cases of simultaneity with `xa`,`xb`,... or `xm`. Another expression for the same result would then be:

```
[6207]      (trg prospects (:xa:) till lim whenno lim)
            opb next (trg prospects (:xb:) till lim whenno lim)
            ...
            opm next (trg prospects (:xm:) till lim whenno lim)
            opn next trg prospects (:xn:) till lim
            heed {H,lim}
```

Remember that **`prospects`** and **`till`** are stronger than selectors.

6.3-More about prospectives: the `TRIG` symbol, recursion and prospected blocks

Dispatches of the trigger may be represented in the object expression by the `TRIG` symbol, similar in some way to the `GO` symbol introduced in 3.5 to represent startup (except that `TRIG` may occur in the time-window it opens) . Dispatches of the trigger expression occurring outside the windows opened are not eliminated, so that the following equivalence holds:

```
[6301]      trg prospects (:TRIG:) till lim  ==  trg
```

`TRIG` may first appear as a term of the object expression with the meaning:

```
[6302]      trg prospects (:TRIG orelse xnotrig:) till lim
                 ==  trg  orelse  trg prospects (:xnotrig:) till lim
```

In this example, `TRIG` is only used to avoid writing the expression `trg` a second time.

Alone or in an **`orelse`** subexpression, `TRIG` may further appear as the initial selectand of a succession, or as the trigger of a secondary prospection (see 6.5 below); for example,

```
[6303]      trg prospects
               (:TRIG opa next Ya ... opm next Ym opn next Yn heed H:)
            till lim
```

where `trg` is a typed message expression. `TRIG` belongs to the referential of the succession, so that only the first dispatch of `Ya` after a dispatch of `trg` is taken into account. The prospective will thus only occur once during the time-window, ***unless the trigger occurs again*** before the limiter.

Such extra dispatches are involved when `TRIG` appears in other positions, such as in

```
[6304]    trg prospects (:Y next TRIG:) till lim
```

or in `[6309]` below.

`TRIG` may be combined with a constant to form a typed message (see 3.5):

```
[6305]    trg prospects (:{TRIG; 'Hi!'} orelse xnotrig:) till lim
```

A prospective may contain the name of the message which it defines, within the prospected expression as well as within trigger and limiter expressions. For example, to convert dispatches of the signal `S` occurring in the time-window into dispatches of a message `M` carrying the ordinal number of each occurrence, one may write the definiton

```
[6306]  M : trg prospects (:(M when next S + 1) orelse {S;1}:) till lim
```

whose second term occurs first. If `M` has already occurred, its preceding dispatch is delayed and incremented; otherwise its token is `S` and it carries the number `1`. Other forms make use of the `TRIG` symbol; for example:

```
[6307]  M : trg prospects (:TRIG then {S;1} orelse (M when next S +1):)
                till lim
```

or

```
[6308]  M : trg prospects (:({TRIG;0} orelse M) + next {S;1}:) till lim
```

If defined as follows, `M` counts the dispatches of `TRIG` itself, instead of those of `S`:

```
[6309]  M : trg prospects (:(M when next TRIG + 1) orelse {TRIG;1}:)
                till lim
```

Otherwise, the result of a prospective may be given a name locally: a definition thus replaces the object expression between "`(:`" and "`:)`". In this way, `[6102]` may be re-written:

```
        (GO orelse OPEN) prospects
  (:OPEN : Cp then Cq then Cr then Cs heed {C0,C1,C2,C3,C4,C5,C6,C7,C8,C9}:)
                   till OPEN
```

Names defined in a block may be used outside.

Auxiliary definitions may be annexed to the prospected expression or main definition. For example, in an other form of `[6307]`:

```
[6310]  M : trg prospects
    (:1Xa orelse 1Xb | 1Xa : TRIG then {S;1} | 1Xb : M when next S + 1:)
        till lim
```

where intermediate messages `1Xa` and `1Xb` are internally defined, or in

```
[6311]      trg prospects
    (:M : 1Xa orelse 1Xb | 1Xa : TRIG then {S;1} | 1Xb : M when next S + 1:)
        till lim
```

where `M` itself is internally defined.

The right-hand side expressions of the internal definitions are subject to the prospection. An object of prospection which includes defintions is a ***block.*** Names defined in a block may appear outside.

Definitions of which the right-hand side expressions are prospectives with identical triggers and limiters may be grouped together in a single prospective, and vice versa:

```
[6312]      M1 : trg prospects (:obj1:) till lim |
            M2 : trg prospects (:obj2:) till lim
         == trg prospects (:M1 : obj1 | M2 : obj2:) till lim
```

A prospective such as the above or that in `[6311]`, whose object is a block of definitions including only named expressions, is not a message expression: it must not be used as a subexpression, nor as the right part in a definition.

6.4 -Transforming prospectives

When a prospective is encountered in a specification during the latter's transformation into a model, the expression `lim` of the limiter (which is a message of any kind) is first replaced by its token's expression **`any`** `lim`**.** The rules defined in chapter 4 are then applied to its trigger `trg`, its object `obj` and the limiter signal **`any`** `lim`. In this way their model expressions are obtained:

```
                    trg  ->  {trg't; trg'c}
                    (:obj:)  ->  (:{obj't; obj'c}:)
                    any lim  ->  lim't
```

The result is a ***model prospective***. Four intermediate states are then defined:

```
    [6401]       {TRIG; TRIG_c} : {trg't; trg'c}
    [6402]       HALT : lim't
    [6403]       WINDO : TRIG after (TRIG or HALT)
```

The Boolean state `WINDO` will be `TRUE` in any time window opened by `trg` and `FALSE` otherwise.

The model expression `{obj't; obj'c}` between "`(:`" and "`:)`" brackets ***is further transformed*** by the following special rules:

The first rule concerns the tokens `T` (now states) of those of its components which are not defined in the object:

```
    [6404]       T  ->  T and WINDO
```

It expresses their prospection on an individual basis. This rule suffers an exception in the case of the symbol `TRIG`, whose dispatches are not subject to prospection:

```
    [6405]       TRIG  ->  TRIG
```

Let `{T; C}` be the expression of a message within the object expression after this step. The second rule eliminates all effects within a particular time-window of those dispatches of the components which occurred previously. To achieve this, the results of the memorisation of the tokens (always by referential markers) are set to `FALSE` by `HALT`, which also prevents memorising them:

```
    [6406]       T after S  ->  (T and not HALT) after (S or HALT)
```

There is an exception for the trigger's token, which may be memorised until the end of the window it opens, even when in coincidence with `HALT`. The appropriate rule is:

```
    [6407]       TRIG after S  ->  TRIG after (S or HALT)
```

Expressions such as

```
                    (T or TRIG) after S
```

will be developed according to `[4702]` before transformation.

The contents of a message need not be memorised when its token is not, so that the following optional rule may be used for contents, except in the case of `TRIG_c`:

```
    [6408]       C after T  ->  C after (T and not HALT)
```

A third rule eliminates the "`trg` **`prospects`** `(:`" and "`:)` **`till`** `lim`" parts of the notation. For an object expression consisting of a simple typed message `Y`, the above three rules reduce to the following single rule:

```
    [6409]       trg prospects (:Y:) till lim  ->  {Y and WINDO; Y_c}
```

The right-hand side expressions of definitions internal to an object block are treated exactly as are object expressions. Consider for example the transformation of `[6311]`. The normal rules `[4408]`, `[4411]` and `[4401]` produce the semi-transformed block:

```
                   (:{M; M_c} : {1Xa or 1Xb; 1Xa_c if 1Xa else 1Xb_c} |
                    {1Xa; 1Xa_c} : {TRIG after (TRIG or S) and S; 1} |
                    {1Xb; 1Xb_c} : {M after (M or S) and S; M_c after M + 1}:)
```

The special rules applied to right-hand side parts then provide transformed defintions to which defintions of `TRIG`, `HALT`, `WINDO` and of the intermediary `S_W` (which is the result of the prospection of `S`) will be added to provide the ***transformed block*** :

```
TRIG : trg |
{M; M_c} : {1Xa or 1Xb; 1Xa_c if 1Xa else 1Xb_c} |
{1Xa; 1Xa_c} : {TRIG after (TRIG or S_W or HALT) and S_W; 1} |
{1Xb; 1Xb_c} :
    {(M and not HALT) after (M or S_W or HALT) and S_W;
     M_c after M + 1} |
S_W : S and WINDO |
WINDO : TRIG after (TRIG or HALT) | HALT : lim
```

6.5 -Multiple prospectives

In the specification, several prospectives may possibly compose an expression.

When a prospective contains other prospectives, the limiter of the outermost prospective limits implicitly the others:

```
[6501]     trg1 prospects (:
            ...trg2 prospects (:object:) till lim2...
            :) till lim1
```

is equivalent to

```
[6502]     trg1 prospects (:
            ... trg2 prospects (:object:) till {lim2,lim1} ...
            :) till lim1
```

The limiter may implicitely be `NEVER`

```
           trg prospects (:object:)  ==  trg prospects (:object:) till NEVER
```

whose actual limiter is that of the containing prospective, if any. Otherwise, the prospection terminates at machine shutdown or if the program is aborted.

The limiter may well be defined in the object block, as in:

```
          GO prospects (:...stop:...:) till stop
```

Whatever the relative positions of multiple prospectives, a corresponding number of sets of symbols `TRIG`, `HALT` and `WINDO` will be required in the model; numerical suffixes will be used here to distinguish between them.

Transformation of `[6501]` will start with the innermost prospective. Rule `[6406]` will be first applied to expressions such as `T after S` of the `(:object:)` model block, then a second time when transforming the object of the prospective encompassing it, with the result

```
[6503]      (T and not HALT2 and not HALT1) after (S or HALT2 or HALT1)
```

which may be seen to correspond to the formulation of the inner prospective in `[6502]`; the effective limiter is thus `HALT2 or HALT1`.

When a secondary prospection is triggered explicitly by a `TRIG` internal to the object expression, as in

```
[6504]  M : trg1 prospects (:
            ...(TRIG orelse M) prospects (:object:) till lim2 ...
            :) till lim1
```

its windows will open in simultaneity with those of the primary prospection. No adaptation of the transformation rules is necessary. Especially:

```
[6505]   trg prospects (:TRIG prospects (:object:) till li2:) till li1
                  ==    trg prospects (:object:) till {li2,li1}
```

6.6 -Prospecting functions

Any message expression or block to be subject to prospection may be included in a separate ***function declaration*** such as

```
                F1 :: (:body:)
```
to which may be added a ***local limiter*** as in:
```
                F2 :: (:body:) till loclim
```
A prospective whose object is a function name (not bracketed by "`(:`" and "`:)`") is then a ***function call*** expression:
```
    [6601]      trg prospects F1 till lim
                  ==  trg prospects (:body:) till lim
```
and
```
    [6602]      trg prospects F2 till lim
                  ==  trg prospects (:body:) till {loclim,lim}
```
A function call "occurs" as does any message expression whenever one of its dispatches occurs.

In object expressions, according to `[6205]` and `[6505]`:
```
    [6603]       F  ==  (:TRIG prospects F:)
    [6604]       (F1 isel F2) == (:TRIG prospects F1 isel TRIG prospects F2:)
    [6605]       F1 prospects F2  ==  (:(TRIG prospects F1) prospects F2:)
```
where ***`isel`*** is an instantaneous selector.

Optional formal parameters may appear in the left-hand side of the declaration, and then in the body of the right-hand side in the rôle of messages or constants:
```
                F(PMa,PMb) :: (:... PMb ... PMa ... PMb ...:)
```
In a call of the above function, the formal parameters `PMa` and `PMb` are replaced by their argument expressions so that
```
                Y : trg prospects F(X,K)
```
is equivalent to
```
                PMa : X | PMb : K |
                Y : trg prospects (:... PMb ... PMa ... PMb ...:)
```
Components of the argument expressions `X` and `K` are thus not individually prospected by `trg`. The parameters may also appear in the local limiter expression.

In the body as well as in the local limiter, the "`$`" symbol represents the result of the prospection, i.e. the dispatches resulting from a call of the function. As an example, consider the declaration of a function derived from `[6307]` which numbers the dispatches of its argument `flop`:
```
    [6606]   order(flop) :: (:TRIG then {flop;1} orelse ($ when next flop +1):)
```
which can then be called to allot successive numbers to the dispatches of `beat`:
```
                trg prospects order(beat)
```
This expression can be indexed (see 2.4) by the value `60` in order to wait for the sixtieth dispatch of `beat` after the trigger:
```
    [6607]      (trg prospects order(beat))[60]
```
A function call may appear in the declaration of a function, as in
```
    [6608]      count(times,flop) :: (:(TRIG prospects order(flop))[times]:)
```
or simply
```
                count(times,flop) :: (:order(flop)[times]:)
```
the following call of which
```
                trg prospects count(60,beat)
```
is equivalent to `[6607]`. If `beat` is the name of the output signal of a one second clock, this prospective will occur one minute after being triggered whenever the time interval between dispatches of `trg` is long enough.

6.7 -Transforming declarations and calls

The method for transforming a specification which includes prospectives into a model was indicated in 6.4 above. Transformation of prospecting functions may be approached in two different ways.

In the first approach, they are considered to be simple formal contrivances with respect to specifications: according to 6.6, function calls are transformed into plain prospectives by defining their parameters with the corresponding arguments and replacing the name of the function by a copy of the function body (and if required merging both limiters). Further transformation into a program then presents no specific features.

In the second approach, function declarations and calls are maintained in the model. The arguments of the calls together with the bodies of the declarations are transformed by the rules from chapter 4, in order to generate ***model function calls*** and ***model function declarations***. Consider the function declaration:

```
[6701]     totalize(toAdd) :: (:{TRIG; 0} orelse $ + next toAdd:)
```

such that the "totalizator" specified in 3.5 may be rewritten:

```
        Accum : GO prospects totalize((Num1 + Num2) orelse Num1 orelse Num2)
```

After this step, the model function declaration (inspired from the model in 4.6) will be

```
[6702]     totalize(TRIG,,OUTRHLT,toAdd,toAdd_c) ::
             $ : TRIG or $ after ($ or toAdd_W or HALT) and toAdd_W |
             $_c : 0 if TRIG else $_c after $ + toAdd_c |
             toAdd_W : toAdd and WINDO |
             WINDO : TRIG after (TRIG or HALT) |
             HALT : OUTRHLT
```

where tokens and contents appear. If used in the object body, `TRIG_c` should appear after `TRIG` in the parameter list.

The model definition of `Accum` will then be

```
[6703]     {Accum; Accum_c} :
             totalize(GO,,,Num1 or Num2,Num1_c+Num2_c if Num1 and Num2
                                              else Num1_c if Num1
                                              else Num2_c)
```

6.8 -Declarations and calls in programs

Basically, in the second approach described in 6.7, the result of the translation of a prospective function declared in the model will consist in segments adapted to be invoked from the different parts of the calling program, entitled (see chapter 5):

```
{initialisation:}
{"tick" part:}
{"tock" part:}
{acquisition:}
```

Conventional programming languages allow this only through the declaration of as many functions. Data communication between them will require a common storage space, of which unrestricted use implies random dynamic allocation. Storage allocation and release requests will be located in called functions, so that actual needs may be inaccessible to the calling programs.

Taking this into account, a tentative sequential program for `[6703]` will be:

```
    {initialisation part}
GO := TRUE;
Num1, Num2 := FALSE;
ini_totalize(Accum_ptr));
repeat
    {"tick" part}
  Sum:=Num1 or Num2;
  if Sum then begin Sum_c:=Num1_c + Num2_c if Num1 and Num2
                                   else Num1_c if Num1
                                   else Num2_c end;
  tick_totalize(Accum_ptr,GO,,FALSE,Sum,Sum_c,Accum,Accum_c);
    {"tock" part}
  dispatch Accum;
  tock_totalize(Accum_ptr);
    {acquisition:}
  GO := FALSE;
  accept Num1, Num2;
  acq_totalize(Accum_ptr)
endlessly
```

Argument `Accum_ptr` is the name of a pointer to the dynamic space used for computing successive values of the variables `Accum` and `Accum_c`. The functions invoked correspond to the model function [6702]; they would be declared under the headers:

```
ini_totalize(var blockptr));
tick_totalize(blockptr,TRIG,TRIG_c,OUTRHLT,toAdd,toAdd_c;var);
tock_totalize(blockptr);
acq_totalize(blockptr);
```

The body of the model function will have been treated in a way similar to that exposed in 5.2 to 5.5, except that variables are elements of a structure, and the resulting sequences distributed in the different functions' bodies.

The function `ini_totalize` first gets a storage block for the structure of local variables, including `WINDO` and `HALT`, then initialises its elements and delivers `blockptr` pointing to it. The function `tick_totalize` computes `$` and `$_c` from the values of actual parameters, and from initial or computed values of local states. It stores values to be used later, with an exception for `a_$`, `a_$` and `a_WINDO`, which `tock_totalize` updates. The dynamic storage space is released by the function `acq_totalize`, at the end of the cycle in which `HALT` becomes `TRUE`.

Chapter 7: Rationale

There exists a need for non-procedural languages to improve the reliability of computer programs.

The first domain for a non-procedural language is that of formal specification.

Any application is required to exhibit some specific behaviour with respect to various kinds of peripheral devices, clocks, files, data bases, which only communicate from time to time: the rôle of the program upon being launched may thus be thought of as being to construct and deliver messages in response to those it perceives.

To specify a program is to indicate which effects are expected from its execution in relation to the causes involved. To be efficient, and also to remain within the range of understanding of interested parties who may well have no particular knowledge of information technology, it is necessary to stay within this basic definition, and in particular to express the expected effects without reference to any technological means of achieving them. Causes and effects are thus considered to be messages of no significant duration: such fugitive causes, either immediate or dating back to some preceding time, will combine to produce each effect.

The operations defined in chapter 2, which we have denoted as "selections" in order to avoid possible confusion, apply to messages and are sufficient to describe all messages which can be derived from them. Most of them are clearly inspired by the traditional Boolean operations; however one additional "selection" had to be constructed in order to combine non-simultaneous messages.

Once a specification language has been set up on this basis, it will constrain the author of a specification to express everything which should be expressed, enforcing the effort to leave nothing indistinct; the author will generally be led into defining intermediate messages which will appear both as effects and intermediate causes in further selections. Everything which is not specific to non-simultaneous message combinations is said to be "instantaneous", such as in the first place operations and banal functions applying to objects of the usual types, however functions programmed to perform any transformations of any objects may be invoked.

The idea that to start to imagine states is to start to design the product has led us to banish any trace of the notion of state from specifications. On the other hand, the notion of states is at the heart of the "model", which is indeed the non-procedural description of an abstract state machine which may be decomposed into "operators" and which is guaranteed to satisfy the specification. The "operations" formalised in the model will call to mind the principal building blocks for sequential logic circuits, which are indeed sufficient to provide the desired reactive behaviour: Boolean and arithmetic components, multiplexers, and data/load registers. These registers implement the abstract **`after`** operators. Chapter 4 shows how the "selections" of the specification may be mechanically transformed or decomposed into the model's "operations".

Due to the very nature of the model's "operators", deriving a program from the model is then essentially the same as simulating a sequential logic circuit while ignoring the system clock. The only new problem is introduced by the dynamic memory allocation required by prospecting functions to allow their unrestricted use. This apart, the reader may consider that chapter 5 is just the description of a commonplace simulation process.

Various declarations will need to be incorporated into the specification to enable its transformation. If some anomaly is detected in the course of the development process, pertinent marginal cases may have been neglected; this will be corrected exclusively by making changes to the specification. The ultimate validity of the program derives from that of the specification itself, and also of course from the validity of the transformation mechanisms used. Additional functions which may possibly be defined in the non-procedural state expression language used for the model, or functions defined directly in the algorithmic language used for the program will need to be validated separately, if they are not already known to be correct.

An experimental tool has been written, which can carry out the transformation of specifications formulated with the use of type declarations. A sample source formulation and the derived object model may be found in Appendix B, sections B.5 and B.6.

Appendix A: Step-by-step recognition

The use of prospecting functions will be considered below as applied to deciding, as characters are received one at a time, if the corresponding character string or ***text*** conforms to a given grammar. The non-procedural nature of the grammar will be preserved in the formal specification of the decision process.

A.1 -The recognition functions

Consider a predefined character string and its allowable variants, and let us assume that its successive characters are received at a single port. Let us then consider how such a text may be recognised incrementally by making use of all the useful information brought by each character as it is received, but without memorising these characters.

We will assume that the allowable variants are defined by a "grammar" made up of BNF rules. In this approach, the primary rule defines the allowable strings as a sequence of symbols from the alphabet of characters and of secondary symbols. Secondary symbols are then defined in the same way, possibly with further secondary symbols.

Let us show how to specify parallel prospection of the different variants. We will assume for the purposes of the argument that the alphabet of the text is restricted to lower case letters, and that a signal among `Sa`... `Sz` occurs when a dispatch of a message `Phi` of the appropriate type is perceived (see 2.4). One of its constituent signals

```
Sa : Phi['a']
....
Sz : Phi['z']
```

occurs each time a particular character arrives. For each one, a ***character function*** is declared, which is a prospecting function (see 6.6) whose call expressions occur each time the corresponding character is the first to be received after a dispatch of the trigger:

```
a :: (:Sa:) till Phi
...
z :: (:Sz:) till Phi
```

Let `Valid` be a signal following which characters are expected to be received. No grammar is needed to recognise a string consisting of a single character. The signal

```
Valid_z : Valid prospects z
```

occurs each time the character `'z'` is the first to be received after a dispatch of `Valid`. This may be generalised by seeking to transform the primary rule of the grammar into a prospective `P` such that the ***success signal***

```
Valid_P : Valid prospects P
```

occurs whenever one of the allowable variants is recognised.

A.2 -String functions and alternative functions

Character functions may be thought of as recognising degenerate strings consisting only of a single character. In the case of a two character string such as `'ca'`, the ***string function***

```
C :: (:c prospects a:)
```

where the right-hand side consists in two functions linked by **`prospects`** so that

```
Valid prospects C
```

expands into a cascade of two character function calls. A dispatch of `C` occurs each time that, immediately after the call of `C` is triggered, `'c'` arrives; it triggers a call of `a`, immediately after which `'a'` arrives: the function `C` recognises the string `'ca'`. The above declaration of `C` may be compared to the traditional BNF rule defining the secondary symbol `C` by:

```
<C> ::= c a
```

For a string of ***n*** characters, the right-hand side of the corresponding string function declaration is a sequence of ***n*** character functions linked by **`prospects`** symbols.

A text with variants may be defined by an ***alternative*** which enumerates the allowable strings. The functions which recognise them, or ***alternative functions***, will include the **`orelse`** selector. In order to accept the strings `'ab'` or `'cab'`, and according to `[6604]` and `[6605]`, the alternative function `D` declared would be

```
D ::(:a prospects b orelse (c prospects a) prospects b:)
```

which defines ***two parallel prospections***, with two sequences of functions separated by **`orelse`** operating in parallel when a call of `D` is triggered. A dispatch of `D` will occur when one or the other of these sequences succeeds. The BNF rule would be

```
<D> ::= a b | c a b
```

In this example, recognition of one of the strings excludes recognition of the other. On the other hand, in the case of the variants `'ca'` and `'cab'` predefined by

```
<G> ::= c a | c a b
```

the function would be

```
G ::(:c prospects a orelse (c prospects a) prospects b:)
```

When triggered just before receiving the string `'cab'`, `G` ***will deliver two dispatches***: one at the end of `'ca'`, and one at the end of `'cab'` in such a case where the longer of the two strings is obtained by extending the shorter of the two.

This property may be generalised to alternatives with several terms, which may then deliver as many dispatches as they contain terms each time they are triggered.

The term ***formation*** will from now on be used to denote a string, possibly reduced to a single character, or an alternative (some of which as we have seen may be recognised several times when starting from a given character in the text), and also by extension the BNF rules defining them. The functions `C`, `D` and `G` may thus be said to recognise the formations `<C>`, `<D>` and `<G>` (which are also secondary symbols of the grammar).

As secondary symbols, `<C>`, `<D>` and `<G>` may be used in place of symbols of the alphabet to define chains of formations, or alternatives of such chains, which are also formations

```
<D> ::=  a b | <C> b
<G> ::= <C> | <C> b
<E> ::= i <C> <G> s
```

The first character of a chained formation follows immediately the last character of the preceding formation.

In a similar way, the functions `C`, `D` and `G` may be used as character functions in function declarations:

```
D :: (:a prospects b orelse C prospects b:)
G :: (:C orelse C prospects b:)
E :: (:((i prospects C) prospects G) prospects s:)
```

which may clearly be obtained ***mechanically*** from the BNF rules: right-hand sides are bracketed by "`(:`" and "`:)`", **`prospects`** symbols are inserted between chained formations, **`orelse`** symbols replace "`|`" symbols, and secondary symbols are replaced by corresponding function names.

In accordance with the definition of chained formations, the right-hand side sequences prospect each formation as soon as the preceding one is recognised by triggering the appropriate function. Alternative functions `D` and `G` carry out parallel prospections of the chained formations in the corresponding BNF rules.

Since `<G>` is an alternative, the secondary symbol `<E>` represents one or the other of two strings. If `G` occurs twice, the character function `s` will be triggered twice. The primary function itself may be triggered repeatedly, without waiting for the completion of ongoing prospections, in particular by

```
Valid : GO orelse any Phi
```

which occurs first at each startup (see 3.5) and then at each character which arrives.

The dispatch of a string function or of an alternative signals success, possibly repetitive, in recognising the corresponding formation.

A simple preprocessor will transform any BNF grammar, rule by rule, into prospecting functions recognising the texts it defines. When the function corresponding to the primary rule is triggered, it will trigger the prospection of the different variants, no preference being given to any particular one.

A.3 -Optional and/or repetitive formations

The definitions of strings and of alternatives are sufficient to allow the definition of any variants of the primary formation, if all variants include at least one character. However, some awkwardness may at times be avoided if ***optional*** intermediate formations are defined as possibly reducing to the empty string of zero length.

The function `F` will be said to prospect the optional character `'c'` if the success signal

```
          Valid_F : Valid prospects F
```

occurs once simultaneously with `Valid` and then also if the next character received is `'c'`:

```
          Valid_F : Valid orelse Valid prospects c
```

The function declaration will be

```
          F :: (:TRIG orelse c:)
```

In accordance with `[6302]`, the first term of `F` is thus the signal triggering the function call, so that `F` occurs at the instant it is triggered and at which the prospection of the second term starts. It could be said that `TRIG` prospects the empty string which is present everywhere in the incoming text: success is thus guaranteed. The corresponding BNF notation is often

```
          <F> ::= [c]
```

In the same way that `<F>` or `[c]` may appear in a BNF rule, the function `F` or the right-hand side of its declaration may be inserted in the expression of a formation. For example, the function `D` from A.2 will thus have the equivalent more concise declaration:

```
          D :: (:((TRIG orelse c) prospects a) prospects b:)
```

corresponding to the BNF notation

```
          <D> ::= [c] a b
```

An optional formation may appear anywhere in a string. For example, the function `G` from A.2 may be declared by

```
          G :: (:(c prospects a) prospects (TRIG orelse b):)
```

corresponding to the BNF notation

```
          <G> ::= c a [b]
```

It will not be necessary to use a declaration of a function which calls itself to indicate the ***repetition*** of a formation: local recursion, using the `$` symbol (see 6.6) in the right-hand side of the declaration will suffice. For example, the formation defined in BNF by

```
          <RF> ::= <F> | <RF><F>
```

will be recognised by

```
          RF :: (:F orelse $ prospects F:)
```

in which the prospection of `F` starts immediately, and then restarts at each dispatch of the success signal `$`.

As is the case for simple BNF rules, rules defining optional or repetitive formations may be transformed mechanically into prospecting functions.

A recognition program will be obtained according to 6.7 and 6.8 from the primary rule and from the declaration of character functions.

Appendix B: the Reflex Game's formal specification and model

The informal specification is taken from a publication by G. Berry and G. Gonthier ("The Esterel Synchronous Programming Language: Design, Semantics, Implementation" [1992]), with the minimum changes required to introduce words (in Courier font) related to its formulation while taking care not to modify the game itself.

B.1 -The Reflex Game specification

The player controls the machine by three means: putting a coin in a coin slot, to start the game; pressing a ready button, to start a reflex time measurement sequence; pressing a stop button, to end a measurement.

The machine reacts by operating the following devices: a numerical display that shows reflex times; a go lamp that signals the beginning of a measurement; a game_over lamp that signals game end; a red lamp that signals that the player has tried to cheat or has abandoned the game; a bell that rings when the player hits the wrong button.

When the machine is turned on, the display shows nothing, the game_over lamp is on, the go and red lamps are off. Each game is composed of a fixed round_number of rounds. The player starts a game by inserting a coin; the game_over lamp turns off and the first round is released. The player presses the ready button; then, after a random amount of time, the go lamp turns on and a measurement starts. The player must react as fast as possible; when the stop button is pressed, the round_ends, the display shows the reflex_time measured in milliseconds (ms) since start, the go lamp turns off and the next round is released. When the last_round is completed, the average reflex time (i.e. the total of the reflex_times measured, divided by round_number) is displayed after a pause of pause_length milliseconds, and the game_over lamp is turned on.

There are six cases of anomaly. Two of them are simple mistakes and make the bell ring:

* the player presses stop instead of ready to start a sequence;

* the player presses ready more than once during a round.

In three other cases, the game stops short, the red and game_over lamps are turned on, the go lamp is turned off:

* the player does not press the ready button within limit_time milliseconds expected after the game_over or go lamp turns off (it is assumed that the game has been abandoned);

* the player does not press the stop button within limit_time milliseconds after the go lamp turns on (the game is also assumed to be abandoned);

* the player presses the stop button after the ready button but before the machine turns the go light on, or at the same time that this happens (this is a cheat!).

The last case is the insertion of a coin during a game. The current game is stopped short, the game_over lamp and, if necessary, the go and red lamps turn off, and a new game is started at once.

In the formulation, names denote

- input signals: coin, ms, ready, stop
- an output signal: bell
- typed output messages
  - alphanumerical: display
  - Boolean: game_over, go, red
- specified numbers: limit_time, pause_length, round_number
- Boolean values: off, on
- intermediate signals: short, cheat, game_end, last_round, release, round_end, start
- intermediate typed messages
  - numerical: average, reflex_time, total

- instantaneous standard functions
  alphanumerical: `decimal`
  numerical: `random`
- declared prospecting functions delivering
  a signal: `count`
  a message of numerical type: `totalize`

The only names not derived from the text of the informal specification are `game_end`, `decimal`, `count` and `totalize`. The prospecting function (see 6.7 and 6.8 in chapter 6 above) `totalize(toAdd)`produces a numerical message whose first dispatch occurs carrying the value `0` at the instant it is called. Later dispatches are simultaneous with those of the message `toAdd`; the value carried at each occurrence is the total to date of the values carried by dispatches of `toAdd` which have occurred since the function call. This running total is only reset to zero by a new dispatch of the trigger. The function `count(number,flop)`, at `[6608]` in chapter 6, produces a signal which occurs in simultaneity with the `number`-th dispatch of `flop` following a call.

B.2 -A first-pass formulation ignoring anomalies

```
average : total / round_number |

display : {GO; ''} orelse decimal(reflex_time orelse average) |

game_end : last_round prospects count(pause_length, ms) |

game_over : {coin; off} orelse {GO orelse game_end; on} |

go : {GO orelse stop; off} orelse {start; on} |

last_round : coin prospects count(round_number, round_end) |

red : {GO; off} |

reflex_time : start prospects (:totalize({ms; 1}) when next round_end:) |

release : coin orelse round_end whenno last_round |

round_end : start then stop |

start : (release then ready) prospects count(random, ms) |

total : coin prospects (:totalize(reflex_time) when next game_end:)
```

At this point, no account has yet been taken of those parts of the informal specification text relative to anomalies.

`release` (see below) launches a round, the first phase of which is formalised by the definition of the `start` signal, occurring `random` ms after the first dispatch of `ready` following `release`. The standard instantaneous `random` function delivers a number as the first argument of `count`; the `ms` signal occurs every millisecond. The second and last phase ends with a dispatch of `round_end`, at the first dispatch of `stop` which follows `start`.

`reflex_time` is a typed message which occurs simultaneously with `round_end`, and which carries the number of dispatches of `ms` between the instants of the dispatches of `start` and of `round_end` excluded.

The `last_round` signal occurs at the `round_number`-th dispatch of `round_end` starting from the dispatch of `coin`.

release accompanies dispatches of coin and then of round_end up to but excluding last_round; it marks the beginning of a round.

The time delay of pause_length ms before game_end is achieved by a call of count by last_round with the arguments pause_length and ms.

total accumulates the values carried by dispatches of reflex_time from coin to game_end, at which instant it delivers the resulting total.

average is the instantaneous product of the division of total by the constant round_number; it occurs at the instant of total, which is also the instant of game_end.

display, game_over and go are update messages of the appropriate types; the GO signal occurring at startup installs the initial display configuration. decimal converts a number to a displayable string.

B.3 -A formulation detecting and flagging anomalies

```
      average : total / round_number |

-->   bell : release prospects (:stop:) till ready
             orelse ready prospects (:ready:) till stop |

-->   cheat : ready then stop heed start |

      display : {GO; ''} orelse decimal(reflex_time orelse average) |

      game_end : last_round prospects count(pause_length, ms) |

      game_over : {coin; off} orelse {GO orelse game_end orelse short; on} |

      go : {GO orelse stop orelse short; off} orelse {start; on} |

      last_round : coin prospects count(round_number, round_end) |

      red : {GO orelse coin; off} orelse {short; on} |

      reflex_time : start prospects (:totalize({ms; 1}) when next round_end:) |

      release : coin orelse round_end whenno last_round |

      round_end : start then stop |

-->   short : coin orelse cheat
            orelse release prospects count(limit_time+1, ms) till ready
            orelse start prospects count(limit_time+1, ms) till stop |

      start : (release then ready) prospects count(random, ms)

      total : coin prospects (:totalize(reflex_time) when next game_end:)
```

In addition to the previous definitions, we now have those of short, which implies that of the intermediate cheat, and of bell.

If in the course of a round a coin is inserted, a cheat is detected or the round is abandoned, short occurs; cheat occurs if stop occurs between ready and start, and also in the limit case where stop is simultaneous with start. Abandonment of the round is detected by timeout whenever count is not stopped in time by ready if it was started by release, or is not stopped in time by stop if it was started by start.

In accordance with the specification text, bell indicates inputs of stop and ready which are tolerated within a round; this can occur any number of times for each of them. The first line indicates the stop inputs tolerated between release and ready, the second line the ready inputs tolerated between ready and stop.

The definitions of game_over, go and of red have been completed. The conflict between coin and short in the definitions of game_over and of red is resolved by the preference given by **orelse** to its first term.

B.4 -A formulation including all effects of anomalies

```
      display : {GO; ''} orelse decimal(reflex_time orelse average) |

      game_over : {coin; off} orelse {GO orelse game_end orelse short; on} |

      go : {GO orelse stop orelse short; off} orelse {start; on} |

      red : {GO orelse coin; off} orelse {short; on} |

-->   coin prospects (:
        average : total / round_number |

        bell : release prospects (:stop:) till ready
               orelse ready prospects (:ready:) till stop |

        cheat : ready then stop heed start |

        game_end : last_round prospects count(pause_length, ms) |

-->     last_round : TRIG prospects count(round_number, round_end) |

        reflex_time : start prospects (:totalize({ms;1}) when next round_end:) |

-->     release : TRIG orelse round_end whenno last_round |

        round_end : start then stop |

-->     short : TRIG orelse cheat
             orelse release prospects count(limit_time+1, ms) till ready
             orelse start prospects count(limit_time+1, ms) till stop |

        start : (release then ready) prospects count(random, ms) |

-->     total : TRIG prospects (:totalize(reflex_time) when next game_end:)

-->                  :) till short
```

With the exception of those messages driving the display configuration, and in order that each game should be independent of its predecessor, all the definitions are prospected by coin and limited by short, including the definition of short itself. Within this prospected block, coin is represented by the symbol TRIG. Since limitation of a prospection is not instantaneous, restarting a game is not inhibited if coin and short coincide.

Development of the formulation was carried on at the same time as that of the informal specification; the latter remains a necessity, being the only source attributing meaning to the messages.

Types will now be introduced to allow for easy transformation into the model. Declarations of prospecting functions derived from [6608], [6606] and [6701] have been added

B.5 -A formulation including type specification

```
                         SPECIFICATION reflex_game

SIGNAL coin, ms, ready, stop |
ALPHA FUNCTION decimal(NUMERICAL) |
NUMERICAL FUNCTION random |
NUMERICAL VALUE limit_time, pause_length, round_number |
BOOLEAN VALUE off, on |

ALPHA MESSAGE display : {GO; ''} orelse decimal(reflex_time orelse average) |
NUMERICAL MESSAGE game_over :
  {coin; off} orelse {GO orelse game_end orelse short; on} |
BOOLEAN MESSAGE go : {GO orelse stop orelse short; off} orelse {start; on} |
BOOLEAN MESSAGE red : {GO orelse coin; off} orelse {short; on} |

coin prospects (:
  NUMERICAL MESSAGE average : total / round_number |
  SIGNAL bell : release prospects (:stop:) till ready
         orelse ready prospects (:ready:) till stop |
  SIGNAL cheat : ready then stop heed start |
  SIGNAL game_end : last_round prospects count(pause_length, ms) |
  SIGNAL last_round : TRIG prospects count(round_number, round_end) |
  NUMERICAL MESSAGE reflex_time :
    start prospects (:totalize({ms;1}) when next round_end:) |
  SIGNAL release : TRIG orelse round_end whenno last_round |
  SIGNAL round_end : start then stop |
  SIGNAL short : TRIG orelse cheat
       orelse release prospects count(limit_time+1, ms) till ready
       orelse start prospects count(limit_time+1, ms) till stop |
  SIGNAL start : (release then ready) prospects count(random, ms) |
  NUMERICAL MESSAGE total :
    TRIG prospects (:totalize(reflex_time) when next game_end:)
              :) till short |

  PFUNCTION count(NUMERICAL VALUE times; SIGNAL flop) SIGNAL TRIGGERED ::
    (:order(flop)[times]:) |

  NUMERICAL PFUNCTION order(SIGNAL flop) SIGNAL TRIGGERED ::
    (:TRIG then {flop; 1} orelse ($ when next flop + 1):) |

  NUMERICAL PFUNCTION totalize(NUMERICAL MESSAGE toAdd) SIGNAL TRIGGERED ::
    (:{TRIG; 0} orelse $ +next toAdd:)
```

The symbols ALPHA and PFUNCTION stand respectively for “string” and “prospecting function”.

Transforming the above specification according to chapters 4 and 6 yields the corresponding model, where types appropriate to abstract state machines appear instead of SIGNAL and MESSAGE. The symbol IBSTATE stands there for “initialised Boolean state”.

B.6 -The model

```
                                MODEL reflex_game

 IBSTATE coin, ms, ready, stop |
 ALPHA FUNCTION decimal(NUMERICAL) |
 NUMERICAL FUNCTION random |
 BOOLEAN VALUE off, on |
 NUMERICAL VALUE limit_time, pause_length, round_number |
 IBSTATE display : GO or reflex_time or average |
 ALPHA STATE display_c : '' if GO else decimal(reflex_time_c if reflex_time else
average_c) |
 IBSTATE game_over : coin or GO or game_end or short |
 NUMERICAL STATE game_over_c : off if coin else on |
 IBSTATE go : GO or stop or short or start |
 BOOLEAN STATE go_c : off if GO or stop or short else on |
 IBSTATE red : GO or coin or short |
 BOOLEAN STATE red_c : off if GO or coin else on |

 IBSTATE TRIGa : coin |
 IBSTATE average : total |
 NUMERICAL STATE average_c : total_c / round_number |
 IBSTATE bell : stop_Wb or ready_Wc |
 IBSTATE cheat : (ready_Wa and not HALTa) after (ready_Wa or stop_Wa or start or
HALTa) and stop_Wa |
 IBSTATE game_end : _pfr2 |
 IBSTATE last_round : _pfr3 |
 IBSTATE reflex_time : (_pfr4 and not HALTd) after (_pfr4 or round_end_Wd or
HALTd) and round_end_Wd |
 NUMERICAL STATE reflex_time_c : _pfr4_c after _pfr4 |
 IBSTATE release : TRIGa or round_end and not last_round |
 IBSTATE round_end : (start and not HALTa) after (start or stop_Wa or HALTa) and
stop_Wa |
 IBSTATE short : TRIGa or cheat or _pfr7 or _pfr8 |
 IBSTATE start : _pfr10 |
 IBSTATE total : (_pfr11 and not HALTe) after (_pfr11 or game_end_We or HALTe)
and game_end_We |
 NUMERICAL STATE total_c : _pfr11_c after _pfr11 |
   IBSTATE ready_Wa : ready and WINDOa |
   IBSTATE stop_Wa : stop and WINDOa |
   IBSTATE ms_Wa : ms and WINDOa |
   IBSTATE _pfr2 : count(last_round,,HALTa,pause_length,ms_Wa) |
   IBSTATE _pfr3 : count(TRIGa,,HALTa,round_number,round_end) |
   IBSTATE _pfr7 : count(release,,ready_Wa or HALTa,limit_time + 1,ms_Wa) |
   IBSTATE _pfr8 : count(start,,stop_Wa or HALTa,limit_time + 1,ms_Wa) |
   IBSTATE _pfr10 : count((release and not HALTa) after (release or ready_Wa or
HALTa) and ready_Wa,,HALTa,random,ms_Wa) |
   IBSTATE WINDOa : TRIGa after (TRIGa or HALTa) |
   IBSTATE HALTa : short |
 IBSTATE TRIGb : release |
   IBSTATE stop_Wb : stop and WINDOb |
   IBSTATE WINDOb : TRIGb after (TRIGb or HALTb) |
   IBSTATE HALTb : HALTa or ready_Wa |

 IBSTATEIBSTATE TRIGc : ready_Wa |
   IBSTATE ready_Wc : ready and WINDOc |
   IBSTATE WINDOc : TRIGc after (TRIGc or HALTc) |
   IBSTATE HALTc : HALTa or stop_Wa | IBSTATE TRIGd : start |

 IBSTATE TRIGd : start |
   IBSTATE ms_Wd : ms and WINDOd |
   {IBSTATE _pfr4; NUMERICAL STATE _pfr4_c} : totalize(TRIGd,,HALTd,ms_Wd,1) |
   IBSTATE round_end_Wd : round_end and WINDOd |
   IBSTATE WINDOd : TRIGd after (TRIGd or HALTd) |
   IBSTATE HALTd : HALTa |

 IBSTATE TRIGe : TRIGa |
   IBSTATE reflex_time_We : reflex_time and WINDOe |
   {IBSTATE _pfr11; NUMERICAL STATE _pfr11_c} : totalize(TRIGe,,HALTe,
reflex_time_We,reflex_time_c) |
   IBSTATE game_end_We : game_end and WINDOe |
   IBSTATE WINDOe : TRIGe after (TRIGe or HALTe) |
   IBSTATE HALTe : HALTa |

 PFUNCTION count(TRIG,,OUTRHLT,times,flop) ::
 IBSTATE flop, TRIG, OUTRHLT |
 NUMERICAL VALUE times |
 IBSTATE $ : _pfr13 and (_pfr13_c = times) |
   IBSTATE flop_W : flop and WINDO |
   {IBSTATE _pfr13; NUMERICAL STATE _pfr13_c} : order(TRIG,,HALT,flop_W) |
   IBSTATE WINDO : TRIG after (TRIG or HALT) |
   IBSTATE HALT : OUTRHLT |

 PFUNCTION order(TRIG,,OUTRHLT,flop) ::
 IBSTATE flop, TRIG, OUTRHLT |
 IBSTATE $ : TRIG after (TRIG or flop_W or HALT) and flop_W or ($ and not HALT)
after ($ or flop_W or HALT) and flop_W |
 NUMERICAL STATE $_c : 1 if TRIG after (TRIG or flop_W or HALT) and flop_W else
$_c after $ + 1 |
   IBSTATE flop_W : flop and WINDO |
   IBSTATE WINDO : TRIG after (TRIG or HALT) |
   IBSTATE HALT : OUTRHLT |

 PFUNCTION totalize(TRIG,,OUTRHLT,toAdd,toAdd_c) ::
 IBSTATE toAdd, TRIG, OUTRHLT |
 NUMERICAL STATE toAdd_c |
 IBSTATE $ : TRIG or ($ and not HALT) after ($ or toAdd_W or HALT) and toAdd_W |
 NUMERICAL STATE $_c : 0 if TRIG else $_c after $ + toAdd_c |
   IBSTATE toAdd_W : toAdd and WINDO |
   IBSTATE WINDO : TRIG after (TRIG or HALT) |
   IBSTATE HALT : OUTRHLT |
```

This model was produced by an experimental program interpreting formal transformation rules. When necessary, `TRIG`, `HALT` and `WINDO` have been indexed with lower case letters rather than with numbers. Since a name must not be defined more than once, its type takes its natural place in front of its definition; bare declarations only correspond to external names, those which are not defined in the specification itself. For clarity, auxiliary definitions are shifted to the right, and groups of lines derived from a prospective or from a prospecting function declaration appear betwen blank lines. However, disregarding function definitions, the separation into groups is meaningless, since the model is not a sequential program. Similarly, the order of definitions is also meaningless.

END