

## Actions and Behaviour

Let us start to introduce some modelling language features which will allow us to model the behaviour of a cell component.

Suppose the cell component holds a single piece of information which is indistinguishable from any other similar piece. That is, we have a data type containing a single element, as opposed to a Boolean (contains *two* elements) or integer (contains *infinite* elements) data type.

The significance of the cell behaviour is whether it *contains* the information or not, rather than what the information is. Later we will use this to model areas of roadway where a car is present or not. This is a *cellular* automata approach to traffic flow modelling and simulation.

This simple example will allow us to highlight our approach to communication without the need to consider data issues. These will be covered later.

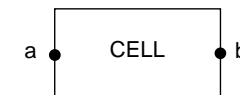
## DEFINITION OF CELL BEHAVIOUR

$$\begin{aligned} \text{CELL} &= a \text{ CELL}' \\ \text{CELL}' &= b \text{ CELL} \end{aligned}$$

In the above example we define the identifier CELL as having the *behaviour* given by the expression on the right hand side of the = symbol, namely expression  $a \text{ CELL}'$ . This behaviour has the intent of performing action  $a$  and then proceeds with the behaviour described by identifier CELL'.

CELL' has its behaviour defined similarly, again using the = symbol which binds an expression (on the right hand side) to the identifying name (on the left hand side)

*Structurally*, the CELL component looks like the following:



The above approach, which uses two distinct process state identifiers, is really just a descriptive convenience. The behaviour of CELL could have been described by a single *process* expression rather than using two mutually recursive expressions, as below:

$$\text{CELL} = a b \text{ CELL}$$

As discussed previously, we do not distinguish input from output. Hence it is a matter of convention that action *a* represents the “input” of the singleton data item, and action *b* the “output”. Assuming that the *process* identified by CELL starts empty then we can assume that *a* corresponds to input and *b* to output. The above definition can be read as follows:

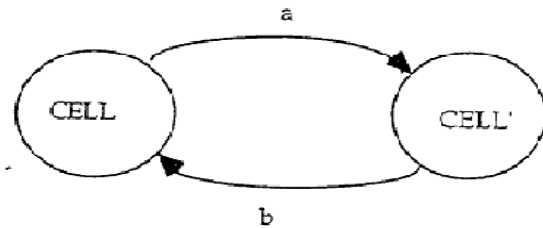
“input data by an *a* action (on the port of the same name) and subsequently output this data by a *b* action (an action on the *b* port) and then repeat indefinitely”.

We can see that the action names are used to identify activities occurring at particular ports with a particular intended meaning. Also, the CELL component has potentially *infinite behaviour*, alternating between an input and an output action. Infinite behaviour is a common feature of many concurrent systems. For example, most digital hardware will continue to function provided components do not burn out and the power is not switched off.

Notice that in the above we have introduced two quite different concepts; *structure* and *behaviour*.

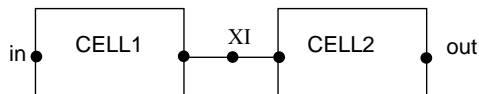
## State Diagrams

An alternative and equivalent way of describing the behaviour of our cell component is as a two state, *finite state automata*. This may be pictured by the following state diagram.



CELL and CELL' again denote the two states of the system with the occurrence of action *transition a* taking us from state CELL to the next state, CELL'. Similarly, action *b* takes us from CELL' to its single successor state CELL. The arrows picture the appropriately *labelled transition* (or actions) taking us from one state to another.

Notice that this state diagram refers to the two *states* that a single cell component may evolve into in contrast to the structural picture of a *two-cell system* involving components CELL1 and CELL2, given by:



## Cash Dispenser Example

The Circal modelling language is used to *describe processes* which have *state*, and which perform, or respond to, *actions*. As an introductory explanation of these terms, we model a cash-dispensing machine using the *state diagram* of Figure 1:

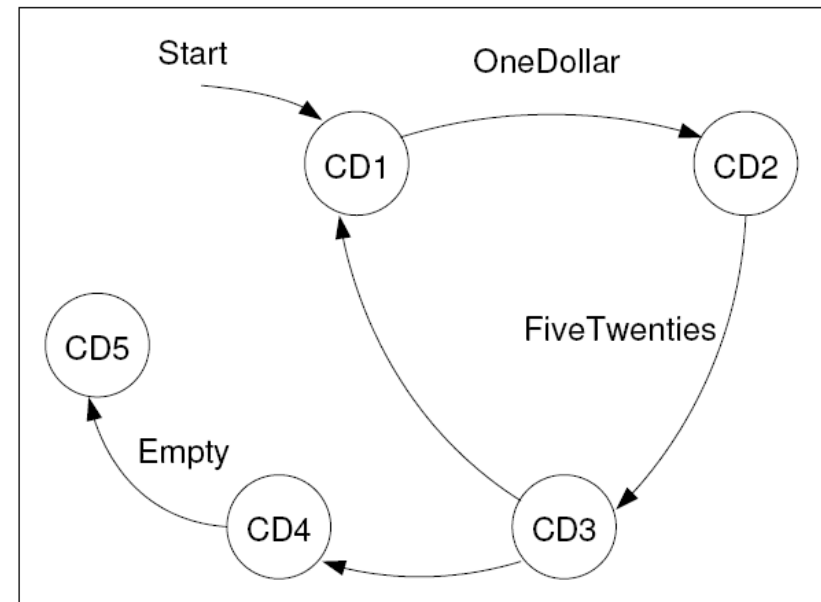


Figure 1: a Cash-Dispenser

Remember that in a state diagram we use circles to denote states, and labelled arrows to denote actions. An unlabelled arrow is known as an *empty transition*, while start states are indicated with a thicker arrow. Terminal states, if any, are those with no arrows leaving them.

In the cash-dispenser example, the actions *OneDollar*, *FiveTwenties* and *Empty* are abstractions of the following “real” activities:

*OneDollar*: A one dollar coin is pushed into the machine, and the machine keeps it. Note that this constitutes a two sided “agreement”; we use *OneDollar* for the situation that the coin is both provided and accepted.

*FiveTwenties*: Five twenty cent pieces are dispensed from the machine.

*Empty*: The machine illuminates its empty indicator.

The states CD1, CD2, CD3, CD4 and CD5 correspond to the following “real” situations:

CD1 The machine has coins to dispense.

CD2 The machine has just accepted a dollar coin.

CD3 The machine has just dispensed five twenty cent coins, and must now determine whether it has any more twenty cent coins left.

CD4 The machine has run out of twenty cent coins, and so must now illuminate its empty indicator.

CD5 The empty indicator is now on, and the machine will participate in no further actions.

The transition from CD3 to CD1 and from CD3 to CD4 are interesting in that they have no associated actions; this results from the fact that state CD3 corresponds to an *internal* state of the machine where the availability of twenty cent coins is checked, i.e. an *external observer* of the machine is not aware of this checking process until the *Empty* action is performed and coins are no longer accepted.

Now that we have a machine, we model a user as in Figure 2.

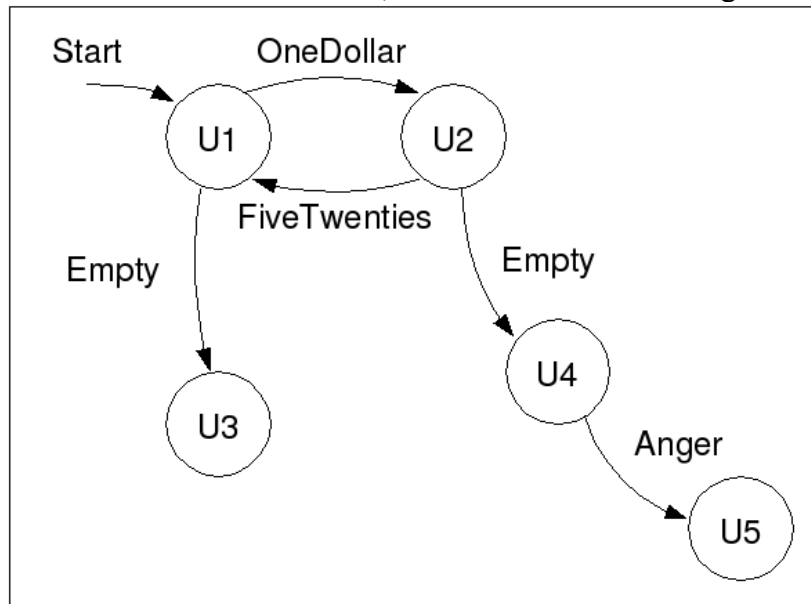


Figure 2: A Cash-Dispenser User

The user continues to put dollar coins into the machine until it is empty. Should the machine respond to a dollar coin by not issuing five twenties, but becomes empty, the user will become angry! We may be interested in asking the following types of questions:

- (1) *what is the composite behaviour of the cash dispenser and the user?*
- (2) *will the user ever become angry?*
- (3) *if so, how can we redesign dispenser to avoid this situation?*

The composite behaviour of CD1 and U1 is very hard to determine just by placing the two diagrams side by side and trying to correlate the actions by eye. This is because we have two *loci of control*, and no obvious way to reconcile the *many potential interactions* between the processes; we need some *rules* to determine what happens when several processes interact. It is the role of interacting automata theories (of which Circal is an example), to define the rules that enable us to answer these questions.

With this short example, a number of important issues have been raised which we now review:

- Actions are used to model arbitrary real world events.
- Processes are used to model sequences of actions (sometimes infinite sequences), often corresponding to a physical entity such as a machine or a program.
- An action can model something which *happens* to a process, or is *initiated by* a process (it is unnecessary to make a distinction).
- An action may be shared between several processes, indicating a potential communication or interaction between them. In such a system of composed communicating processes, a shared action often has a *happens to* interpretation in one process and a *initiated by* interpretation in an other.

## Modelling

We have previously introduced two features which require representation within a modelling formalism: *structure* and *behaviour*.

Structural description concerns itself with denoting the interconnection of components e.g. within a computer network. Due to the structural complexity found in many systems, techniques for modelling structure which mirrors any natural *hierarchy* in the system are required.

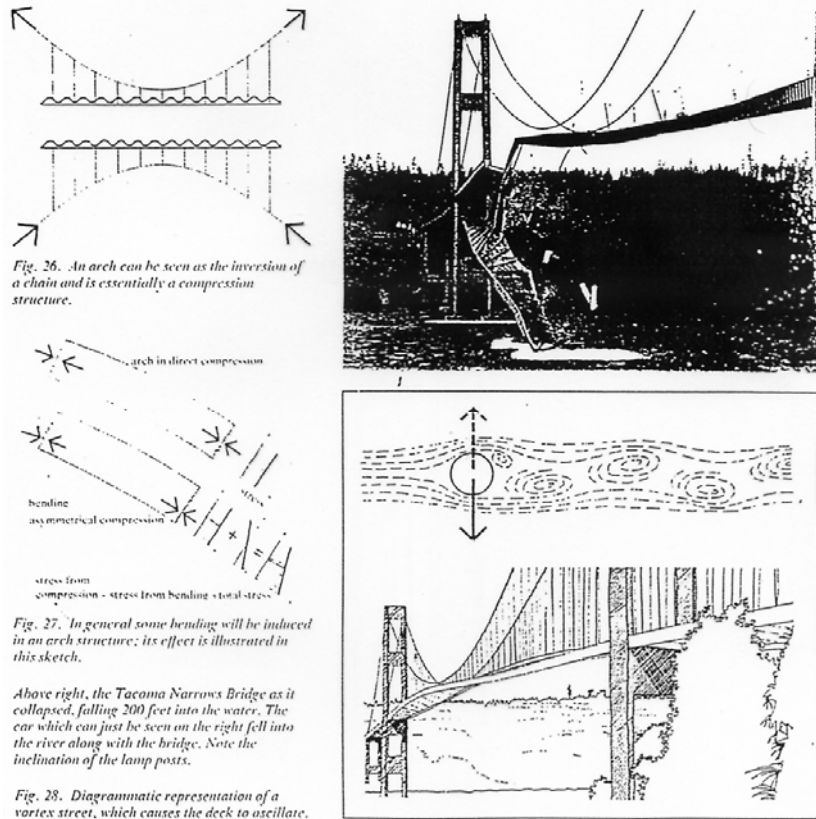
Behavioural description is fundamental to a modelling formalism, since it is the behaviour of the system that we are required to analyse and validate. A good system design will usually have a hierarchical structure; if the behavioural description can capitalise on this structure then the descriptive task will be simplified.

## System Design

We use a *divide and conquer* philosophy for structuring the descriptive task. We describe a complex, concurrent system by a suitable combination of the (simpler) descriptions of the (simpler) component parts. Hence when we program, we construct complex software from many smaller inter-connected component programs.

Both behaviour and structure need to be expressed using a *description language*. In the electronics world they are called HDL's, hardware description languages. Software specification languages such as Z and UML also exist.

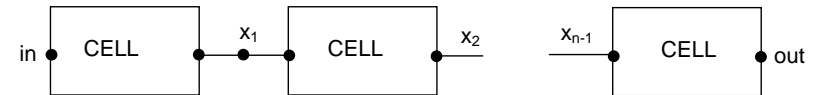
Consider modelling a bridge. We must capture *dynamic* properties of the structure rather than just the physical structure. Why model (describe) then analyse? Self evident from the following example of the Tecoma Narrows Bridge.



## Modelling a Constructed System

### A Buffer Example

Given the previous description of our CELL component, which alternately takes a unary data item in one port and then passes it out on the other, how do we construct a buffer from a number of them?



Using a binary operator (between *two* operands) we can connect two CELL processes together by joining the output of one to the input of another. In fact, we assume that similarly named ports are joined by the *composition operator*. This *structural operator* can be used to describe the "wiring together" of ports from each of two components. In fact it may wire together more than one pair of ports but again each wiring must involve a *unique* port, one from each component.

Let us use the symbol \* (called star) between the two components to denote that they are *potentially* concurrently active and that they *may* interact or communicate. Then we have

$$\underbrace{\text{CELL} * \text{CELL} * \dots * \text{CELL}}_{n \text{ times}}$$

as the description (in our prototype formalism) of the above n cell buffer.

Given the behaviour of a single CELL is specified as:

$$\begin{aligned} \text{CELL} &= a \text{ CELL}' \\ \text{CELL}' &= b \text{ CELL} \end{aligned}$$

then if each of the component cells start in the state CELL (as opposed to CELL') then they each correspond to an empty storage cell.

What then is the behaviour of

$$\underbrace{\text{CELL} * \text{CELL} * \dots * \text{CELL}}_{n \text{ times}} ?$$

## Nomenclature

Remembering that a description in our formalism is given by a process description, or *process* for short.

Each process (or automaton to use an equivalent name) corresponds to some physical object or agent, including single components such as the CELL component above.

Just as physical systems are built from simpler components, so process descriptions are built from simpler process descriptions.

Thus the \* operator above is used to *construct* a process description from a number of instances of the simpler CELL process.

This newly constructed process may also be given a unique name since this construction *creates a new process* which models a distinct and particular behaviour.

The *definition* symbol = may also be used for this purpose, *binding* a new process identifier to the process expression.

$$\text{BUFF}_n = \text{CELL} * \text{CELL} * \dots * \text{CELL}$$

## Processes and State

Initially we introduced the idea of the model of an agent being in a *state*, and evolving to a *new state* following the occurrence of an *action*. This action will belong to the (finite) set of possible actions that the model, and the agent which it models, may perform.

CELL and CELL' are *states* of a 2 state process.

The *constructed* process  $\text{BUFF}_n$  was defined in terms of  $n$  individual CELL components. We have not so far indicated whether these components are exact copies of the individual CELL whose behaviour was expressed previously nor have we indicated how this construction is to take place.

## Specification

However, there is another way to describe the behaviour of a buffer of capacity  $n$ , and that is using a direct behavioural *specification* in terms of how the buffer interacts with the *outside environment*, rather than by how it is constructed. In fact, a specification will purposefully avoid giving details of the internal construction of a system, the buffer in this case. This is important as there may be a number of different constructions (or *designs*) of a buffer of capacity  $n$  which have identical behaviour, with respect to how data enters and leaves the entire buffer.

This is in fact the very behaviour which should be described by the specification, with the specification being a more *abstract* description which ignores all *internal* activities such as the communication between adjacent cells in a particular buffer *implementation*.

The *\** operator for *constructing* (or synthesising) the behaviour of the model of a concurrent system from the description of its components *mimics* the interaction or communication between component agents. Together with a suitable *abstraction* operator they form the basis of this approach to concurrent system description design.

## Process Syntax

Let us look at specifying the behaviour of a buffer of size  $n$ .

Suppose we have parameterisable identifier names, then we can define a buffer process called  $\text{BUFF}_n$  parameterised on an integer which indicates how many "units" of information are held. Identifier names can really be anything as they are just *syntactic* objects (i.e. strings of symbols) which will then be *bound* to a particular behaviour - the process (you may wish to think of a process in terms of a finite state diagram). We frequently find that the choice of identifier is useful in the definition procedure, as with this example.

Initially the buffer is empty, information is then input unit by unit until  $n$  are held. At this stage they will be output.

$$\begin{aligned}\text{BUFF}_n(0) &= \text{in } \text{BUFF}_n(1) \\ \text{BUFF}_n(n) &= \text{out } \text{BUFF}_n(n-1) \\ \text{BUFF}_n(i) &= \text{in } \text{BUFF}_n(i+1) + \text{out } \text{BUFF}_n(i-1) \\ &\quad \text{where } 0 < i < n\end{aligned}$$

The first two definitions are self explanatory; if the buffer is empty then the buffer process (in state  $\text{BUFF}_n(0)$ ) can only input data by an *in* action leaving us in state  $\text{BUFF}_n(1)$ . If full then it is in state  $\text{BUFF}_n(n)$  and can only output via an *out* action.

The third alternative is that we are neither full nor empty. In this case we can accept *either* an input from the surrounding environment, *or* we can output to the environment. Notice that it is the environment which controls this activity. The parameterising integer is incremented if a data item is input and is decremented if it is output. It is the *choice operator* denoted by  $+$  which represents the interaction with the environment where a single action is selected to occur.

After one or other (and not both) of the actions *in* or *out* occur, the state of the process will evolve from  $\text{BUFF}_n(i)$  to  $\text{BUFF}_n(i+1)$  or  $\text{BUFF}_n(i-1)$  respectively.

For example, at a certain point in time the environment (that is, other processes modelling other components which will interact with the buffer) may only wish to send to the buffer, at other times it may only wish to accept from it.

Note that  $\text{BUFF}_n(0)$  is just an identifier name; it is *not* a function  $\text{BUFF}_n$  taking argument "0"!

## Simultaneity

But suppose we wish to model the possibility of the environment both sending to and receiving from the buffer *simultaneously*. For example, the *in* port of the buffer is connected to one component, the *out* to another and the independent interactions with the buffer just happen simultaneously. This is represented by an alternative definition of a buffer, call it  $BUF_n(i)$ , now with three alternatives:

$$BUF_n(i) = in BUF_n(i+1) + out BUF_n(i-1) + (in\ out) BUF_n(i)$$

The third *summand* in the definition uses a *composite* action (*in out*) which denotes the two singleton actions *in* and *out* occurring *simultaneously*.

## Describing Variables and Values

The communication of values between processes can be readily described using multiple ports, and hence multiple links. For each element (i.e. a particular value) of a data type we have a separate port which is used to represent that particular value, in either input or output mode.

As an example consider how we model Booleans. Here we have two values 0 and 1. If we wish to communicate on a link named *m* say; then we are required to use two ports named *m0* and *m1*. The former represent the event “send value 0 on link *m*, either in or out” while the latter corresponds to the communication of value 1.

The following example illustrates this approach to modelling data communication:

## Sequencer Example

A Boolean sequencer may wish to input Boolean values on link  $i$  and then output them on  $j$  prior to further potential input. This is defined as follows:

$$\text{SEQ} = i0 \ j0 \ \text{SEQ} + i1 \ j1 \ \text{SEQ}$$

Notice that we use a composite event name such as  $i0$ , to represent the passing (in this case the input) of value  $0$  on link  $i$ .

The event and the port are both named  $i0$ ; this is really just a composite name which *represents* the transmission of a zero on a link called  $i$ .

Notice that it is not a function applied to zero - it is nothing other than a convenient naming convention which we can use to represent a particular physical phenomena.

## Structure

The SEQ process has following structure:



The communication of integers and reals may be effected similarly. Indeed, any data type may be handled in this way. Data types with infinite members (infinite cardinality) will of course require infinite *conceptual* ports with which to realise communication.

## Representing Programming Language Variables

Consider the program fragment in the following example:

### Example

```
bool x,y
y: = 0;
x: = y
```

How do we describe the meaning of these assignment statements using our interacting automata process ideas?

We now know how to describe data communication, which is the basis of assignment, so now we need to concentrate on modelling the variables.

## Locations in Memory

Variables are themselves processes: they correspond to *locations in memory* that are allocated by the declaration:

```
bool x,y
```

These locations may then be accessed and the current value *read* or output; or they may be *updated* with a new value, *overwriting* the previous one. When the variable is used on the left hand side of the assignment symbol, convention tells us that we are going to *write* to the location to update it. When the variable is on the right, we know that we are going to *read* the value and *then* do something with it, such as change it and load the result into another location, again *denoted* by a variable, possibly itself.

What then is the *sequence* of steps involved in the assignment command `y: = 0`?

The basic operation is the communication of the value zero to the location denoted by variable `y`, with zero overwriting any prior value stored in the location.

A location is therefore modelled by a process with two ports; one for writing to it (which we will call a *load* action) and one for reading from it (a *fetch* action). Suppose the variable that is being modelled by this process has the unary data type, as used in our previous cell and buffer examples, then a location is modelled as follows:

$$\begin{aligned} \text{LOC} &= \text{load } \text{LOC}' \\ \text{LOC}' &= \text{fetch } \text{LOC}' + \text{load } \text{LOC}' \end{aligned}$$

The initially empty location requires a load action to “fill” it before a fetch operation may be performed on it. Once loaded (as specified by *new* process  $\text{LOC}'$ ) the *environment* determines whether to fetch the contents or load a new (unary) value into it.

We clearly require a number of locations of any given data type, corresponding to the distinct variables of that type used within the program being modelled. It is therefore convenient to use a notational device, such as *subscripting by* the variable name, to distinguish which location process corresponds to which variable.

Hence for a variable  $x$  of unary type, we may define its behaviour as follows:

$$\begin{aligned} \text{LOC}_x &= \text{load}_x \text{LOC}' \\ \text{LOC}'_x &= \text{fetch}_x \text{LOC}'_x + \text{load}_x \text{LOC}'_x \end{aligned}$$

Notice that the load and fetch *ports* are also indexed by the variable name to distinguish them from load and fetch ports used to communicate data to and from other variables.

This example is rather unrealistic in its use of unary data; the modelling of a boolean variable denoting a location is more illustrative of this approach. As indicated earlier in this chapter, we model the communication of boolean values on a single physical port (called  $m$  say) by two actions ( $m0$  and  $m1$  in this case) occurring on the two ports of the same name.

A boolean variable called  $x$  denotes the following location process:

$$\begin{aligned} \text{BLOC\_INIT}_x &= \text{load}_x 0 \text{BLOC}_x(0) + \text{load}_x 1 \text{BLOC}_x(1) \\ \text{BLOC}_x(0) &= \text{fetch}_x 0 \text{BLOC}_x(0) + \text{load}_x 0 \text{BLOC}_x(0) + \text{load}_x 1 \text{BLOC}_x(1) \\ \text{BLOC}_x(1) &= \text{fetch}_x 1 \text{BLOC}_x(1) + \text{load}_x 0 \text{BLOC}_x(0) + \text{load}_x 1 \text{BLOC}_x(1) \end{aligned}$$

The set of port labels give the structure as follows:

$$\{\text{fetch}_x 1, \text{fetch}_x 0, \text{load}_x 0, \text{load}_x 1\}$$

Notice the use of an initialisation phase of activity to perform the *initial* load operations. We are now almost in a position to describe the example using a number of communicating, concurrently-active processes. What remains is to decide on how to model constants, such as the zero in the example. A constant ZERO process which outputs a succession of zeros is given by:

$$\text{ZERO} = \text{out}0 \text{ ZERO}$$

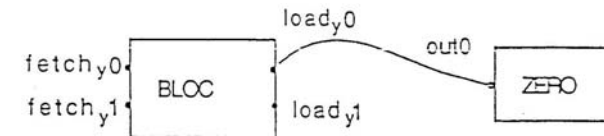
This *recursive* definition gives us a sequence of *out0* events.

Our example will now involve the three processes

$$\text{BLOC}_x, \text{BLOC}_y \text{ and ZERO}$$

Let us first model the assignment statement  $y := 0$ . This involves connecting the single port of the ZERO process, which we called *out0*, to the *load<sub>y</sub>0* port of process  $\text{BLOC}_y$ .

This is pictured as follows:



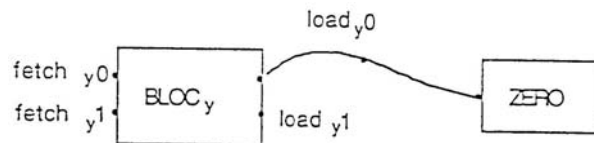
Now the “wiring up” *convention* that we use is such that similarly named ports join together. For the ports known as *load<sub>y</sub>0* and *out0* to be connected requires that they (both or one or the other) be renamed using the same name. A renaming or relabelling operator is provided in Circal for this purpose. It is a postfix operator (occurs *after* the operand) that is used to replace a label in the sort of a process by a new label, distinct from all others in the sort. Let us assume that we can replace port *out0* in process ZERO (its only port in fact) by *load<sub>y</sub>0*.

The interconnected system which models the first assignment using processes  $\text{BLOC}_y$ , and ZERO is now described in Circal as follows, using the composition operator \*:

$$\text{BLOCINIT}_y * \text{ZERO}[\text{load}_y 0 / \text{out}0]$$

Notice the use of the relabelling operator where  $loady0$  replaces  $out0$ .

This system is pictured as follows:



We use the process BLOC\_INIT (or more accurately the BLOC process in *state* BLOC\_INIT) since the assignment  $y := 0$  involves only a load operation, replacing any previous value held in the corresponding location.

The second assignment  $x := y$  may be described using the processes BLOC<sub>x</sub> and BLOC<sub>y</sub> (again) which model the actions occurring on locations (representing variables)  $x$  and  $y$  respectively. This assignment involves fetching the contents of location  $y$  and loading them into location  $x$ . It can be modelled directly by connecting the *fetch* ports of process BLOC<sub>y</sub> to the corresponding (0 to 0 and 1 to 1) *load* ports of BLOC<sub>x</sub>.

## Sequential Location Access

However, a problem with this approach is now apparent. The separate references to location  $y$  in each assignment statement refer to activities on the *same* process BLOC<sub>y</sub> but at separate *points in time*. This separation in time is frequently represented in programming languages by the use of the semicolon as a sequencing construct. Hence, it is not sufficient to just connect our two location processes together using their data communication ports - we must also *control* the occurrence of fetch and load operations.

We are now approaching the realms of microprogramming and the specification of the *control unit* of a (simple) microprocessor.

Rather than specify the inherently complex behaviour of an appropriate control unit, we may instead use a simple controller process which will perform two distinct functions: it will *sequence* successive assignment statements and it will determine whether a *load* or a *fetch* is the appropriate operation to be performed on a location depending on whether the variable is on the left or right of the assignment operator.

## Sequential Controller

For this example, a suitable controller generates the sequence

$$load_y \quad fetch_y \quad load_x.$$

We need not distinguish whether the load or fetch involves a 0 or a 1; only the particular sequence of operations is significant. Furthermore, we also need to determine the routing of the data such that  $fetch_y$  obtains a value that is subsequently loaded into the  $x$  location.

$$\begin{aligned} CTRL &= load_y 0 CTRL' + load_y 1 CTRL' \\ CTRL &= fetch_y 0 load_x 0 \Delta + fetch_y 1 load_x 1 \Delta \end{aligned}$$

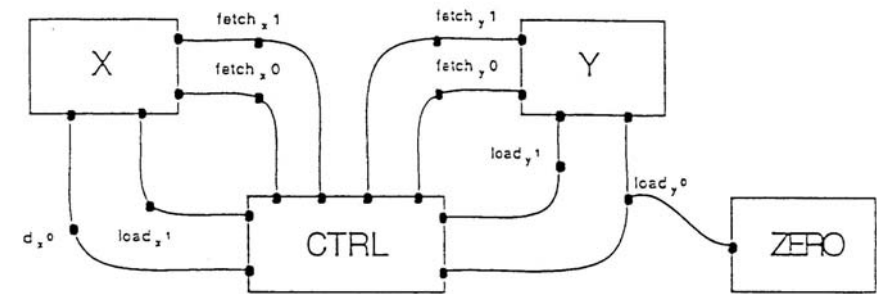
Here process CTRL loads location  $y$ , since variable  $y$  is on the left hand side in the first assignment statement. Process CTRL then evolves to state CTRL' representing the sequential nature of the semicolon. Process CTRL' models the control necessary for the second assignment statement where a fetch *from* location  $y$  is followed by a load *to* location  $x$ , with the communication of boolean values 0 or 1 treated separately. Both these  $fetch_y/load_x$  sequences are terminated by the null process  $\Delta$  indicating *stop*.

## System Structure

The system now consists of four processes, as follows:

$$BLOCINIT_x * BLOCINIT_y * CTRL * ZERO [load_y/out0]$$

The *structure* of this system is pictured as follows:



This diagram indicates where *data flows* between the two locations via, and under the control of, the control process.

Notice also the three-way interaction that occurs between CTRL, the location  $y$  and the ZERO process. The ability of the constant ZERO process to continuously interact on its load<sub>0</sub> port allows it to continue to participate with its two partners, even *after* the initial loading of the constant zero to location  $y$ , as from the first assignment statement.

A *multiway linkage* such as this indicates that *all* of the connected processes must participate in the occurrence of that particular action; that is CTRL, ZERO and the  $y$  location must all be able to perform a load<sub>0</sub> action for that action to occur in the *composite system*. It is the \* composition operator in Circal which effects this *synchronised communication*.

## Circal Modelling Language Features

Before proceeding further, let us reflect on the features of our concurrency modelling language that have been presented so far.

- *Sequentially* is described by the juxtapositioning of actions (on labelled ports) as in  $a b P$ .
- *Choice* of actions, determined by other processes interacting through the corresponding ports (the environment), is described using the sum operator  $+$  as in  $a P + b Q$ ; that is, either event  $a$  or event  $b$  occurs and the process evolves to renewal processes  $P$  or  $Q$  respectively.
- *Terminating* processes are described using the null processes  $\Delta$  as in  $a \Delta$ ; that is, do event  $a$  and then stop.

- *Relabelling* a process to create a particular linkage, and to produce particular instances of a generic process, is described using the “port and event  $y$  replaces port and event  $x$ ” operator, as with  $P[y/x]$ .
- *Recursive processes* can be defined, as with  $P = aP$ , using the definition operator which binds a process with an *identifier name*.
- *Simultaneous actions* are naturally described as with  $(a \ b) P$ . Actions  $a$  and  $b$  occur simultaneously and the process evolves to  $P$ .
- *Concurrent composition* of two (or more) processes is described using the composition operator, as with  $P * Q$ . This operator is both associative and commutative.