1

Processes

In this chapter, processes are introduced as expressions of a simple language built from a few basic operators. The behaviour of a process E is characterised by transitions of the form $E \xrightarrow{a} F$, that E may become F by performing the action a. Structural rules prescribe behaviour, since the transitions of a compound process are determined by those of its components. Concrete pictorial summaries of behaviour are presented as labelled graphs, which are collections of transitions. We review various combinations of processes and their resulting behaviour.

1.1 First examples

A simple process is a clock that perpetually ticks.

$$\texttt{Cl} \overset{\mathrm{def}}{=} \texttt{tick.Cl}$$

Names of actions such as `tick` are in lower case, whereas names of processes such as `Cl` have an initial capital letter. A process definition ties a process name to a process expression. In this case, `Cl` is attached to `tick.Cl`, where both occurrences of `Cl` name the same process. The defining expression for `Cl` invokes a prefix operator . that builds the process $a.E$ from the action a and the process E.

Behaviour of processes is captured by transitions $E \xrightarrow{a} F$, that E may evolve to F by performing or accepting the action a. The behaviour of `Cl` is elementary, since it can only perform `tick` and in so doing becomes `Cl`

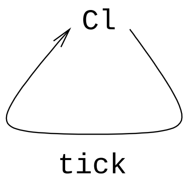


FIGURE 1.1. The transition graph for `Cl`

$$
\begin{array}{lcl}
\texttt{Ven} & \overset{\text{def}}{=} & \texttt{2p}.\texttt{Ven}_{\texttt{b}} + \texttt{1p}.\texttt{Ven}_{\texttt{l}} \\
\texttt{Ven}_{\texttt{b}} & \overset{\text{def}}{=} & \texttt{big}.\texttt{collect}_{\texttt{b}}.\texttt{Ven} \\
\texttt{Ven}_{\texttt{l}} & \overset{\text{def}}{=} & \texttt{little}.\texttt{collect}_{\texttt{l}}.\texttt{Ven}
\end{array}
$$

FIGURE 1.2. A vending machine

again. This is a consequence of the rules for deriving transitions. First is the axiom for the prefix operator.

$$\boxed{\mathrm{R}(.) \quad a.E \xrightarrow{a} E}$$

A process $a.E$ performs the action a and becomes E. An instance of this axiom is the transition $\texttt{tick.Cl} \xrightarrow{\texttt{tick}} \texttt{Cl}$. The next transition rule refers to the operator $\overset{\text{def}}{=}$, and is presented with the desired conclusion uppermost.

$$\boxed{\mathrm{R}(\overset{\text{def}}{=}) \quad \frac{P \xrightarrow{a} F}{E \xrightarrow{a} F} \; P \overset{\text{def}}{=} E}$$

If the transition $E \xrightarrow{a} F$ is derivable and $P \overset{\text{def}}{=} E$, then $P \xrightarrow{a} F$ is also derivable. Goal-directed transition rules are used because we are interested in discovering the available transitions of a process. There is a single transition for the clock, $\texttt{Cl} \xrightarrow{\texttt{tick}} \texttt{Cl}$. Suppose our goal is to derive a transition $\texttt{Cl} \xrightarrow{a} E$. Because the only applicable rule is $\mathrm{R}(\overset{\text{def}}{=})$, the goal reduces to the subgoal $\texttt{tick.Cl} \xrightarrow{a} E$, and the only possibility for deriving this subgoal is an application of R(.), in which case a is `tick` and E is `Cl`.

The behaviour of `Cl` is represented graphically in Figure 1.1. Ingredients of this behaviour graph (known as a "transition system") are process expressions and binary transition relations between them. Each vertex is a process expression, and one of the vertices is the initial vertex `Cl`. Each derivable transition of a vertex is depicted. Transition systems abstract from the derivations of transitions.

An unsophisticated vending machine `Ven` is defined in Figure 1.2. The definition of `Ven` employs the binary choice operator + (which has wider scope than the prefix operator) from Milner's CCS, Calculus of Communicating Systems [42, 44]. Initially `Ven` may accept a `2p` or `1p` coin, and then

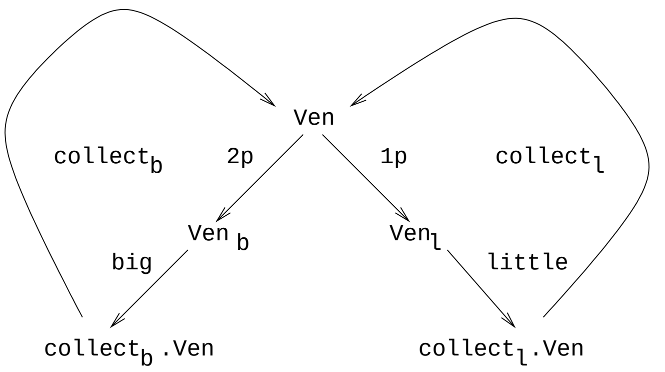


FIGURE 1.3. The transition graph for `Ven`

a button `big` or `little` may be depressed depending on the coin deposited, and finally after an item is collected the process reverts to its initial state. There are two transition rules for +.

$$\boxed{\text{R}(+) \quad \frac{E_1 + E_2 \xrightarrow{a} F}{E_1 \xrightarrow{a} F} \qquad \frac{E_1 + E_2 \xrightarrow{a} F}{E_2 \xrightarrow{a} F}}$$

The derivation of the transition $\texttt{Ven} \xrightarrow{\texttt{2p}} \texttt{Ven}_\texttt{b}$ is as follows.

$$\cfrac{\texttt{Ven} \xrightarrow{\texttt{2p}} \texttt{Ven}_\texttt{b}}{\cfrac{\texttt{2p.Ven}_\texttt{b} + \texttt{1p.Ven}_\texttt{l} \xrightarrow{\texttt{2p}} \texttt{Ven}_\texttt{b}}{\texttt{2p.Ven}_\texttt{b} \xrightarrow{\texttt{2p}} \texttt{Ven}_\texttt{b}}}$$

The goal reduces to the subgoal beneath it as a result of an application of $\text{R}(\overset{\text{def}}{=})$, which in turn reduces to the axiom instance via an application of the first of the R(+) rules. When presenting proofs of transitions, side conditions in the application of a rule, such as $\text{R}(\overset{\text{def}}{=})$, are omitted. Figure 1.3 pictures the transition system for `Ven`.

A transition $E \xrightarrow{a} F$ is an assertion derivable from the rules for transitions. To discover the transitions of E, it suffices to examine its main combinator and the transitions of its components. There is an analogy with rules for expression evaluation. To evaluate $(3 \times 2) + 4$ it suffices to evaluate the components 3×2 and 4, and then sum their values. Such families of rules give rise to a structural operational semantics, as pioneered by Plotkin [49]. However, whereas the essence of an expression is to be evaluated, the essence of a process is to act.

Families of processes can be defined using indexing. A simple case is the set of counters $\{\texttt{Ct}_i : i \in \mathbb{N}\}$ of Figure 1.4. The counter $\texttt{Ct}_3$ can increase to $\texttt{Ct}_4$ by performing `up` or decrease to $\texttt{Ct}_2$ by performing `down`. The derivation of the transition $\texttt{Ct}_3 \xrightarrow{\texttt{up}} \texttt{Ct}_4$ is as follows.

$$\begin{aligned} \mathtt{Ct}_0 &\overset{\mathrm{def}}{=} \mathtt{up.Ct}_1 + \mathtt{round.Ct}_0 \\ \mathtt{Ct}_{\mathtt{i+1}} &\overset{\mathrm{def}}{=} \mathtt{up.Ct}_{\mathtt{i+2}} + \mathtt{down.Ct}_{\mathtt{i}} \end{aligned}$$

FIGURE 1.4. A family of counters

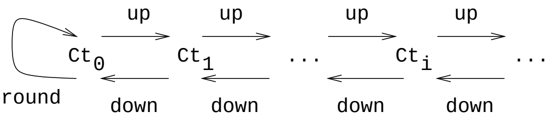


FIGURE 1.5. The transition graph for $\mathtt{Ct}_i$

$$\frac{\dfrac{\mathtt{Ct}_3 \xrightarrow{\mathtt{up}} \mathtt{Ct}_4}{\mathtt{up.Ct}_4 + \mathtt{down.Ct}_2 \xrightarrow{\mathtt{up}} \mathtt{Ct}_4}}{\mathtt{up.Ct}_4 \xrightarrow{\mathtt{up}} \mathtt{Ct}_4}$$

The rule $\mathrm{R}(\overset{\mathrm{def}}{=})$ is here applied to the instance $\mathtt{Ct}_3 \overset{\mathrm{def}}{=} \mathtt{up.Ct}_4 + \mathtt{down.Ct}_2$. Each member $\mathtt{Ct}_{\mathtt{i}}$ determines the same transition graph of Figure 1.5 which contains an infinite number of vertices. This graph is "infinite state" because the behaviour of $\mathtt{Ct}_{\mathtt{i}}$ may progress through any of the processes $\mathtt{Ct}_{\mathtt{j}}$, in contrast to the finite state graphs of Figures 1.1 and 1.3.

The operator $+$ can be extended to indexed families $\sum\{E_i : i \in I\}$ where I is a set of indices. $E_1 + E_2$ abbreviates $\sum\{E_i : i \in \{1, 2\}\}$. Indexed sum may be coupled with indexing of actions. An example is a register storing numbers, represented as a family $\{\mathtt{Reg}'_i : i \in \mathbb{N}\}$.

$$\mathtt{Reg}'_{\mathtt{i}} \overset{\mathrm{def}}{=} \mathtt{read}_{\mathtt{i}}.\mathtt{Reg}'_{\mathtt{i}} + \sum\{\mathtt{write}_j.\mathtt{Reg}'_j : j \in \mathbb{N}\}$$

The act of reading the content of the register when i is stored is $\mathtt{read}_{\mathtt{i}}$, whereas $\mathtt{write}_{\mathtt{j}}$ is the action that updates its value to j. The single transition rule for $\sum$ generalises the rules for $+$.

$$\boxed{\mathrm{R}(\textstyle\sum) \quad \dfrac{\sum\{E_i : i \in I\} \xrightarrow{a} F}{E_j \xrightarrow{a} F} \ j \in I}$$

Consequently, $\mathtt{Reg}'_{\mathtt{i}}$ is able to carry out any $\mathtt{write}_{\mathtt{j}}$ (and thereby changes to $\mathtt{Reg}'_{\mathtt{j}}$) as well as $\mathtt{read}_{\mathtt{i}}$ (and then remains unchanged). A special case is when the indexing set I is empty. By the rule $\mathrm{R}(\sum)$, this process has no transitions, since the subgoal can never be fulfilled. In CCS the nil process $\sum\{E_i : i \in \emptyset\}$ is abbreviated to `0` (and to `STOP` in Hoare's CSP, Communicating Sequential Processes [31]).

Actions can be viewed as ports or channels, means by which processes can interact. It is then also important to consider the passage of data between processes along these channels, or through these ports. In CCS, input of data at a port named a is represented by the prefix $a(x).E$, where $a(x)$ binds free occurrences of x in E. (In CSP $a(x)$ is written $a?x$.) The port label a no longer names a single action, instead it represents the set $\{a(v) \, : \, v \in D\}$ where D is the appropriate family of data values. The transition axiom for this prefix input form is

$$\boxed{\mathrm{R(in)} \quad a(x).E \xrightarrow{a(v)} E\{v/x\} \quad \text{if} \;\; v \in D}$$

where $E\{v/x\}$ is the process term that results from replacing all free occurrences of x in E with v[1]. Output at a port named a is represented in CCS by the prefix $\overline{a}(e).E$ where e is a data expression. The overbar $^{-}$ symbolises output at the named port. (In CSP $\overline{a}(e)$ is written $a!e$.) The transition rule for output depends on extra machinery for expression evaluation. Assume that $\mathrm{Val}(e)$ is the data value in D (if there is one) to which e evaluates.

$$\boxed{\mathrm{R(out)} \quad \overline{a}(e).E \xrightarrow{\overline{a}(v)} E \quad \text{if } \mathrm{Val}(e) = v}$$

The asymmetry between input and output is illustrated by the following process that copies a value from `in` and then sends it through `out`.

$$\texttt{Cop} \stackrel{\mathrm{def}}{=} \texttt{in}(x).\overline{\texttt{out}}(x).\texttt{Cop}$$

Below is a derivation of the transition $\texttt{Cop} \xrightarrow{\texttt{in}(v)} \overline{\texttt{out}}(v).\texttt{Cop}$ for $v \in D$.

$$\frac{\texttt{Cop} \xrightarrow{\texttt{in}(v)} \overline{\texttt{out}}(v).\texttt{Cop}}{\texttt{in}(x).\overline{\texttt{out}}(x).\texttt{Cop} \xrightarrow{\texttt{in}(v)} \overline{\texttt{out}}(v).\texttt{Cop}}$$

The subgoal is an instance of R(in), as $(\overline{\texttt{out}}(x).\texttt{Cop})\{v/x\}$ is $\overline{\texttt{out}}(v).\texttt{Cop}$[2], and so the goal follows by an application of $\mathrm{R}(\stackrel{\mathrm{def}}{=})$. The process $\overline{\texttt{out}}(v).\texttt{Cop}$ has only one transition $\overline{\texttt{out}}(v).\texttt{Cop} \xrightarrow{\overline{\texttt{out}}(v)} \texttt{Cop}$ that is an instance of R(out), since we assume that $\mathrm{Val}(v)$ is v. Whenever `Cop` inputs a value at `in`, it immediately disgorges it through `out`. The size of the transition graph for `Cop` depends on the size of the data domain D, and is finite when D is a finite set.

---

[1] The process $a(x).E$ can be viewed as an abbreviation of the process $\sum\{a_v.E\{v/x\} \, : \, v \in D\}$, writing a_v instead of $a(v)$.

[2] `Cop` contains no free variables because $\texttt{in}(x)$ binds x, and so $(\overline{\texttt{out}}(x).\texttt{Cop})\{v/x\}$ equals $\overline{\texttt{out}}(v).(\texttt{Cop}\{v/x\})$ because x is free in $\overline{\texttt{out}}(x)$, and $(\texttt{Cop}\{v/x\})$ is `Cop`.

Example 1 $\mathtt{Cop_1} \overset{\text{def}}{=} \mathtt{in}(x).\mathtt{in}(x).\overline{\mathtt{out}}(x).\mathtt{Cop_1}$ is a different copier. It takes in two data values at `in`, discarding the first but sending out the second. $\mathtt{Cop_1}$ has initial transition $\mathtt{Cop_1} \xrightarrow{\mathtt{in}(v)} \mathtt{in}(x).\overline{\mathtt{out}}(x).\mathtt{Cop_1}$ for $v \in D$.

Input actions and indexing can be mingled, as in the following redescription of the family of registers, where both i and x have type $\mathbb{N}$.

$$\mathtt{Reg_i} \overset{\text{def}}{=} \overline{\mathtt{read}}(i).\mathtt{Reg_i} + \mathtt{write}(x).\mathtt{Reg}_x$$

$\mathtt{Reg_i}$ can output the value i at the port `read`, or instead it can be updated by being written to at `write`. Below is the derivation of $\mathtt{Reg_5} \xrightarrow{\mathtt{write}(3)} \mathtt{Reg_3}$.

$$\cfrac{\mathtt{Reg_5} \xrightarrow{\mathtt{write}(3)} \mathtt{Reg_3}}{\cfrac{\overline{\mathtt{read}}(5).\mathtt{Reg_5} + \mathtt{write}(x).\mathtt{Reg}_x \xrightarrow{\mathtt{write}(3)} \mathtt{Reg_3}}{\mathtt{write}(x).\mathtt{Reg}_x \xrightarrow{\mathtt{write}(3)} \mathtt{Reg_3}}}$$

The variable x in $\mathtt{write}(x)$ binds the free occurrence of x in $\mathtt{Reg}_x$. An index can also be presented explicitly as a parameter.

Example 2 The multiple copier $\mathtt{Cop}'$ uses the parameterised subprocess $\mathtt{Cop}(n, x)$, where n ranges over $\mathbb{N}$ and x over texts.

$$\begin{array}{lcl} \mathtt{Cop}' & \overset{\text{def}}{=} & \mathtt{no}(n).\mathtt{in}(x).\mathtt{Cop}(n,x) \\ \mathtt{Cop}(0,x) & \overset{\text{def}}{=} & \overline{\mathtt{out}}(x).\mathtt{Cop}' \\ \mathtt{Cop}(i+1,x) & \overset{\text{def}}{=} & \overline{\mathtt{out}}(x).\mathtt{Cop}(i,x) \end{array}$$

The initial transition of $\mathtt{Cop}'$ determines the number of extra copies of a manuscript, for instance $\mathtt{Cop}' \xrightarrow{\mathtt{no}(4)} \mathtt{in}(x).\mathtt{Cop}(4,x)$. The next transition settles on the text, $\mathtt{in}(x).\mathtt{Cop}(4,x) \xrightarrow{\mathtt{in}(v)} \mathtt{Cop}(4,v)$. Then before reverting to the initial state, five copies of v are transmitted through the port out.

Data expressions may involve operations on values, as in the following example, where x and y range over a space of messages.

$$\mathtt{App} \overset{\text{def}}{=} \mathtt{in}(x).\mathtt{in}(y).\overline{\mathtt{out}}(x^\wedge y).\mathtt{App}$$

`App` receives two messages m and n on `in` and transmits their concatenation $m^\wedge n$ on `out`. We shall assume different expression types, such as boolean expressions. An example is that $\mathrm{Val}(\mathit{even}(i)) = \text{true}$ if i is an even integer and is false otherwise. This allows us to use conditionals in the definition of a process as exemplified by `S` that sieves odd and even numbers.

$$\mathtt{S} \overset{\text{def}}{=} \mathtt{in}(x).\textbf{if } \mathit{even}(x) \textbf{ then } \overline{\mathtt{out}}_{\mathtt{e}}(x).\mathtt{S} \textbf{ else } \overline{\mathtt{out}}_{\mathtt{o}}(x).\mathtt{S}$$

Below are the transition rules for the conditional.

$$\boxed{\text{R(if 1)} \quad \frac{\textbf{if } b \textbf{ then } E_1 \textbf{ else } E_2 \xrightarrow{a} E'}{E_1 \xrightarrow{a} E'} \quad \text{Val}(b) = \text{true}}$$

$$\boxed{\text{R(if 2)} \quad \frac{\textbf{if } b \textbf{ then } E_1 \textbf{ else } E_2 \xrightarrow{a} E'}{E_2 \xrightarrow{a} E'} \quad \text{Val}(b) = \text{false}}$$

$\mathtt{S}$ initially receives a numerical value through the port `in`. For instance, $\mathtt{S} \xrightarrow{\mathtt{in}(55)} \textbf{if } \mathit{even}(55) \textbf{ then } \overline{\mathtt{out}_\mathtt{e}}(55).\mathtt{S} \textbf{ else } \overline{\mathtt{out}_\mathtt{o}}(55).\mathtt{S}$. It then outputs through $\mathtt{out}_\mathtt{e}$ if the received value is even, or through $\mathtt{out}_\mathtt{o}$ otherwise. In this example, $\textbf{if } \mathit{even}(55) \textbf{ then } \overline{\mathtt{out}_\mathtt{e}}(55).\mathtt{S} \textbf{ else } \overline{\mathtt{out}_\mathtt{o}}(55).\mathtt{S} \xrightarrow{\overline{\mathtt{out}_\mathtt{o}}(55)} \mathtt{S}$.

Example 3 Consider the following family of processes for $i \geq 1$.

$$\mathtt{T}(i) \overset{\text{def}}{=} \textbf{if } \mathit{even}(i) \textbf{ then } \overline{\mathtt{out}}(i).\mathtt{T}(i/2) \textbf{ else } \overline{\mathtt{out}}(i).\mathtt{T}((3i+1)/2)$$

So $\mathtt{T}(5)$ performs the sequence of transitions

$$\mathtt{T}(5) \xrightarrow{\overline{\mathtt{out}}(5)} \mathtt{T}(8) \xrightarrow{\overline{\mathtt{out}}(8)} \mathtt{T}(4) \xrightarrow{\overline{\mathtt{out}}(4)} \mathtt{T}(2)$$

and then cycles through the transitions $\mathtt{T}(2) \xrightarrow{\overline{\mathtt{out}}(2)} \mathtt{T}(1) \xrightarrow{\overline{\mathtt{out}}(1)} \mathtt{T}(2)$.

Exercises

1. Draw the transition graphs for the following clocks.
    - (a) $\mathtt{Cl}_1 \overset{\text{def}}{=} \mathtt{tick.tock.Cl}_1$
    - (b) $\mathtt{Cl}_2 \overset{\text{def}}{=} \mathtt{tick.tick.Cl}_2$
    - (c) $\mathtt{Cl}_3 \overset{\text{def}}{=} \mathtt{tick.Cl}$
    - (d) `tick.0`
2. Show that there are two derivations of the transition $\mathtt{Cl}_4 \xrightarrow{\mathtt{tick}} \mathtt{Cl}_4$ when $\mathtt{Cl}_4 \overset{\text{def}}{=} \mathtt{tick.Cl}_4 + \mathtt{tick.Cl}_4$. Draw the transition graph for $\mathtt{Cl}_4$.
3. Contrast the behaviour of $\mathtt{Cl}_5 \overset{\text{def}}{=} \mathtt{tick.Cl}_5 + \mathtt{tick.0}$ with that of `Cl` by drawing their transition graphs.
4. Define a more rational vending machine than `Ven` that allows the big button to be pressed if two 1p coins are entered, and the little button to be depressed twice after a 2p coin is deposited.

5. Assume that the space of values consists of two elements, 0 and 1. Draw transition graphs for the following three copiers `Cop`, $\mathtt{Cop}_1$ and $\mathtt{Cop}_2$ where $\mathtt{Cop}_2 \overset{\mathrm{def}}{=} \mathtt{in}(x).\overline{\mathtt{out}}(x).\overline{\mathtt{out}}(x).\mathtt{Cop}_2$.

6. Draw transition graphs of $\mathtt{T}(31)$ and $\mathtt{T}(17)$, where $\mathtt{T}(i)$ is defined in Example 3.

7. For any processes E, F and G, show that the transition graphs for $E + F$ and $F + E$ are isomorphic, and that the transition graph for $(E + F) + G$ is isomorphic to that of $E + (F + G)$.

8. From Walker [60]. Define a process `Change` that describes a change-making machine with one input port and one output port, that is capable initially of accepting either a 20p or a 10p coin, and that can then dispense any sequence of 1p, 2p, 5p and 10p coins, the sum of whose values is equal to that of the coin accepted, before returning to its initial state.

1.2 Concurrent interaction

A compelling feature of process theory is modelling of concurrent interaction. A prevalent approach is to appeal to handshake communication as primitive. At any one time, only two processes may communicate at a port or along a channel. In CCS, the resultant communication is a *completed* internal action. Each incomplete, or observable, action a has a partner $\overline{a}$, its co-action. Moreover, the action $\overline{\overline{a}}$ is a, which means that a is also the co-action of $\overline{a}$. The partner of a parameterised action $\mathtt{in}(v)$ is $\overline{\mathtt{in}}(v)$. Simultaneously performing an action and its co-action produces the internal action τ, which is a complete action that does not have a partner.

Concurrent composition of E and F is expressed as $E \,|\, F$. Below is the crucial transition rule for $|$ that conveys communication.

$$\boxed{\mathrm{R}(\,|\,\mathrm{com}) \qquad \dfrac{E \,|\, F \xrightarrow{\tau} E' \,|\, F'}{E \xrightarrow{a} E' \qquad F \xrightarrow{\overline{a}} F'}}$$

If E can carry out an action and become E', and F can carry out its co-action and become F' then $E \,|\, F$ can perform the completed internal action τ and become $E' \,|\, F'$. Consider a potential user of the copier `Cop` of the previous section, who first writes a file before sending it through the port `in`.

$$\begin{array}{lcl} \mathtt{User} & \overset{\mathrm{def}}{=} & \mathtt{write}(x).\mathtt{User}_x \\ \mathtt{User}_{\mathtt{v}} & \overset{\mathrm{def}}{=} & \overline{\mathtt{in}}(v).\mathtt{User} \end{array}$$

As soon as `User` has written the file v, it becomes the process $\mathtt{User_v}$ that can communicate with `Cop` at the port `in`. Rule R($\mid$com) is used in the following derivation[3] of the transition $\mathtt{Cop} \mid \mathtt{User_v} \xrightarrow{\tau} \overline{\mathtt{out}}(v).\mathtt{Cop} \mid \mathtt{User}$.

$$\dfrac{\mathtt{Cop} \mid \mathtt{User_v} \xrightarrow{\tau} \overline{\mathtt{out}}(v).\mathtt{Cop} \mid \mathtt{User}}{\dfrac{\mathtt{Cop} \xrightarrow{\mathtt{in}(v)} \overline{\mathtt{out}}(v).\mathtt{Cop}}{\mathtt{in}(x).\overline{\mathtt{out}}(x).\mathtt{Cop} \xrightarrow{\mathtt{in}(v)} \overline{\mathtt{out}}(v).\mathtt{Cop}} \quad \dfrac{\mathtt{User_v} \xrightarrow{\overline{\mathtt{in}}(v)} \mathtt{User}}{\overline{\mathtt{in}}(v).\mathtt{User} \xrightarrow{\overline{\mathtt{in}}(v)} \mathtt{User}}}$$

The goal transition is the resultant communication at `in`. Through this communication, the value v is sent from the user to the copier because $\mathtt{User_v}$ performs the output $\overline{\mathtt{in}}(v)$ and `Cop` performs the input $\mathtt{in}(v)$, where they agree on the value v. Data is thereby passed from one process to another. When the actions a and $\overline{a}$ do not involve values, the resulting communication is a synchronization.

Several users can share the copying resource. $\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}})$ involves two users, but only one at a time is allowed to employ it. So, other transition rules for $\mid$ are needed, permitting components to proceed without communicating.

$$\boxed{\mathrm{R}(\mid) \quad \dfrac{E \mid F \xrightarrow{a} E' \mid F}{E \xrightarrow{a} E'} \qquad \dfrac{E \mid F \xrightarrow{a} E \mid F'}{F \xrightarrow{a} F'}}$$

In the first of these rules, the process F does not contribute to the action a that E performs. Below is a sample derivation.

$$\dfrac{\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}}) \xrightarrow{\tau} \overline{\mathtt{out}}(v1).\mathtt{Cop} \mid (\mathtt{User} \mid \mathtt{User_{v2}})}{\dfrac{\mathtt{Cop} \xrightarrow{\mathtt{in}(v1)} \overline{\mathtt{out}}(v1).\mathtt{Cop}}{\mathtt{in}(x).\overline{\mathtt{out}}(x).\mathtt{Cop} \xrightarrow{\mathtt{in}(v1)} \overline{\mathtt{out}}(v1).\mathtt{Cop}} \quad \dfrac{\mathtt{User_{v1}} \mid \mathtt{User_{v2}} \xrightarrow{\overline{\mathtt{in}}(v1)} \mathtt{User} \mid \mathtt{User_{v2}}}{\dfrac{\mathtt{User_{v1}} \xrightarrow{\overline{\mathtt{in}}(v1)} \mathtt{User}}{\overline{\mathtt{in}}(v1).\mathtt{User} \xrightarrow{\overline{\mathtt{in}}(v1)} \mathtt{User}}}}$$

The goal transition reflects a communication between `Cop` and $\mathtt{User_{v1}}$, meaning $\mathtt{User_{v2}}$ is not a contributor. $\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}})$ is not forced to engage in communication. Instead, it may carry out an input action $\mathtt{in}(v)$, or an output action $\overline{\mathtt{in}}(v1)$ or $\overline{\mathtt{in}}(v2)$.

$$\begin{aligned}
&\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}}) \xrightarrow{\mathtt{in}(v)} \overline{\mathtt{out}}(v).\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}}) \\
&\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}}) \xrightarrow{\overline{\mathtt{in}}(v1)} \mathtt{Cop} \mid (\mathtt{User} \mid \mathtt{User_{v2}}) \\
&\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}}) \xrightarrow{\overline{\mathtt{in}}(v2)} \mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User})
\end{aligned}$$

---

[3] We assume that $\mid$ has greater scope than other process operators. The process $\overline{\mathtt{out}}(v).\mathtt{Cop} \mid \mathtt{User}$ is therefore the parallel composition of $\overline{\mathtt{out}}(v).\mathtt{Cop}$ and `User`.

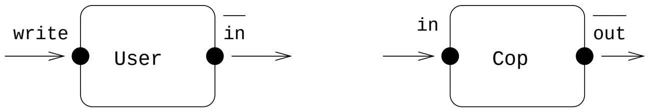


FIGURE 1.6. Flow graphs of `User` and `Cop`

The second of these transitions is derived using two applications of R(|).

$$\begin{array}{c} \mathtt{Cop} \mid (\mathtt{User}_{\mathtt{v1}} \mid \mathtt{User}_{\mathtt{v2}}) \xrightarrow{\overline{\mathtt{in}}(v1)} \mathtt{Cop} \mid (\mathtt{User} \mid \mathtt{User}_{\mathtt{v2}}) \\ \hline \mathtt{User}_{\mathtt{v1}} \mid \mathtt{User}_{\mathtt{v2}} \xrightarrow{\overline{\mathtt{in}}(v1)} \mathtt{User} \mid \mathtt{User}_{\mathtt{v2}} \\ \hline \mathtt{User}_{\mathtt{v1}} \xrightarrow{\overline{\mathtt{in}}(v1)} \mathtt{User} \\ \hline \overline{\mathtt{in}}(v1).\mathtt{User} \xrightarrow{\overline{\mathtt{in}}(v1)} \mathtt{User} \end{array}$$

The behaviour of the users sharing the copier is not impaired by the order of parallel subcomponents, or by placement of brackets. Both processes $(\mathtt{Cop} \mid \mathtt{User}_{v1}) \mid \mathtt{User}_{v2}$ and $\mathtt{User}_{v1} \mid (\mathtt{Cop} \mid \mathtt{User}_{v2})$ have the same capabilities as $\mathtt{Cop} \mid (\mathtt{User}_{v1} \mid \mathtt{User}_{v2})$. These three process expressions have isomorphic transition graphs, and therefore in the sequel we omit brackets between multiple concurrent processes[4].

The parallel operator is expressively powerful. It can be used to describe infinite state systems without invoking infinite indices or value spaces. A simple example is the following counter `Cnt`.

$$\mathtt{Cnt} \overset{\mathrm{def}}{=} \mathtt{up}.(\mathtt{Cnt} \mid \mathtt{down}.0)$$

`Cnt` can perform `up` and become $\mathtt{Cnt} \mid \mathtt{down}.0$ that can perform `down`, or a further `up` and become $\mathtt{Cnt} \mid \mathtt{down}.0 \mid \mathtt{down}.0$, and so on.

Figure 1.6 offers an alternative pictorial representation of the copier `Cop` and user `User`. Such diagrams are called “flow graphs” by Milner [44] (and should be distinguished from transition graphs). A flow graph summarizes the potential movement of information flowing into and out of ports, and also exhibits the ports through which a process is, in principle, willing to communicate. In the case of `User`, the incoming arrow to the port labelled `write` represents input, whereas the outgoing arrow from $\overline{\mathtt{in}}$ symbolises output. Figure 1.7 shows the flow graph for $\mathtt{Cop} \mid \mathtt{User}$ with the crucial feature that there is a potential linkage between the output port `in` of `User` and its input in `Cop`, permitting information to circulate from `User` to `Cop` when communication takes place. However, this port is still available for other users. Both users in $\mathtt{Cop} \mid \mathtt{User} \mid \mathtt{User}$ are able to communicate at different times with `Cop`, as illustrated in Figure 1.8

The situation in which a user has private access to a copier is modelled using an abstraction or encapsulation operator that conceals ports. CCS

---

[4] Equivalences between processes is discussed in Chapter 3.

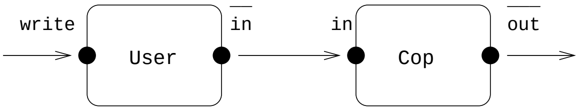


FIGURE 1.7. Flow graph of `Cop` | `User`

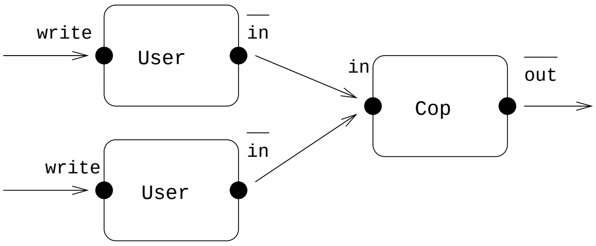


FIGURE 1.8. Flow graph of `Cop` | `User` | `User`

has a *restriction* operator $\backslash J$, where J ranges over families of incomplete actions (thereby excluding the complete action τ). If K is $\{\texttt{in}(v) : v \in D\}$ when D contains the values that can flow through `in`, then the port `in` within (`Cop` | `User`)$\backslash K$ is inaccessible to other users. The flow graph of (`Cop` | `User`)$\backslash K$ is pictured in Figure 1.9, where the linkage without names at the ports represents their concealment from other users, so it can be simplified as in the second diagram of the figure.

The visual effect of $\backslash K$ on the flow graph in Figure 1.9 is justified by the transition rule for restriction, which is as follows where $\overline{J}$ is $\{\overline{a} : a \in J\}$.

$$\boxed{\mathrm{R}(\backslash) \quad \frac{E\backslash J \xrightarrow{a} F\backslash J}{E \xrightarrow{a} F} \; a \notin J \cup \overline{J}}$$

The behaviour of $E\backslash J$ is part of that of E, as any action that $E\backslash J$ may carry out can also be performed by E, but not necessarily the other way round. For instance, `Cop` | `User` is able to perform an `in` input action, whereas an attempt to derive an `in` transition from (`Cop` | `User`)$\backslash K$ is precluded

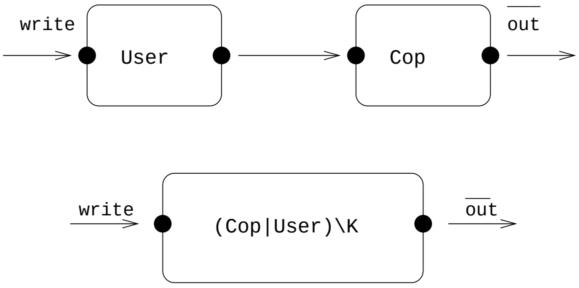


FIGURE 1.9. Flow graph of (`Cop` | `User`)$\backslash K$

$$\begin{array}{lcl}
\texttt{Road} & \overset{\text{def}}{=} & \texttt{car.up.}\overline{\texttt{ccross}}\texttt{.}\overline{\texttt{down}}\texttt{.Road} \\
\texttt{Rail} & \overset{\text{def}}{=} & \texttt{train.green.}\overline{\texttt{tcross}}\texttt{.}\overline{\texttt{red}}\texttt{.Rail} \\
\texttt{Signal} & \overset{\text{def}}{=} & \overline{\texttt{green}}\texttt{.red.Signal} + \overline{\texttt{up}}\texttt{.down.Signal} \\
 & & \\
\texttt{Crossing} & \equiv & (\texttt{Road} \mid \texttt{Rail} \mid \texttt{Signal})\backslash\{\texttt{green}, \texttt{red}, \texttt{up}, \texttt{down}\}
\end{array}$$

FIGURE 1.10. A level crossing

because of the side condition on the rule for R($\backslash$). The presence of $\backslash K$ in $(\texttt{Cop} \mid \texttt{User})\backslash K$ prevents `Cop` from ever doing an `in` transition, except in the context of a communication with `User`. Restriction can therefore be used to enforce communication between parallel components. After the initial write transition $(\texttt{Cop} \mid \texttt{User})\backslash K \xrightarrow{\texttt{write}(v)} (\texttt{Cop} \mid \texttt{User}_{\texttt{v}})\backslash K$, the next transition must be a communication.

$$\dfrac{(\texttt{Cop} \mid \texttt{User}_{\texttt{v}})\backslash K \xrightarrow{\tau} (\overline{\texttt{out}}(v)\texttt{.Cop} \mid \texttt{User})\backslash K}{\dfrac{\texttt{Cop} \mid \texttt{User}_{\texttt{v}} \xrightarrow{\tau} \overline{\texttt{out}}(v)\texttt{.Cop} \mid \texttt{User}}{\dfrac{\texttt{Cop} \xrightarrow{\texttt{in}(v)} \overline{\texttt{out}}(v)\texttt{.Cop}}{\texttt{in}(x)\texttt{.}\overline{\texttt{out}}(x)\texttt{.Cop} \xrightarrow{\texttt{in}(v)} \overline{\texttt{out}}(v)\texttt{.Cop}} \quad \dfrac{\texttt{User}_{\texttt{v}} \xrightarrow{\overline{\texttt{in}}(v)} \texttt{User}}{\overline{\texttt{in}}(v)\texttt{.User} \xrightarrow{\overline{\texttt{in}}(v)} \texttt{User}}}}$$

A port a is concealed by restricting all the actions $\{a(v) : v \in D\}$, and therefore we shall usually abbreviate such a subset within a restriction to $\{a\}$.

Process descriptions can become quite large, especially when they consist of multiple components in parallel. We shall therefore employ abbreviations of process expressions using the relation $\equiv$, where $P \equiv F$ means that P abbreviates F, which is typically a large expression.

Example 1 The mesh of abstraction and concurrency is further revealed in the finite state example without data of a level crossing in Figure 1.10 from Bradfield and the author [10], consisting of three components `Road`, `Rail` and `Signal`. The actions `car` and `train` represent the approach of a car and a train, `up` opens the gates for the car, $\overline{\texttt{ccross}}$ is the car crossing, $\overline{\texttt{down}}$ closes the gates, `green` is the receipt of a green signal by the train, $\overline{\texttt{tcross}}$ is the train crossing, and `red` automatically sets the light red. Unlike most crossings, it keeps the barriers down except when a car actually approaches and tries to cross. The flow graphs of the components, and of the overall system are depicted in Figure 1.11. The transition graph is pictured in Figure 1.12. Both `Road` and `Rail` are simple cyclers that can only perform a determinate sequence of actions repeatedly.

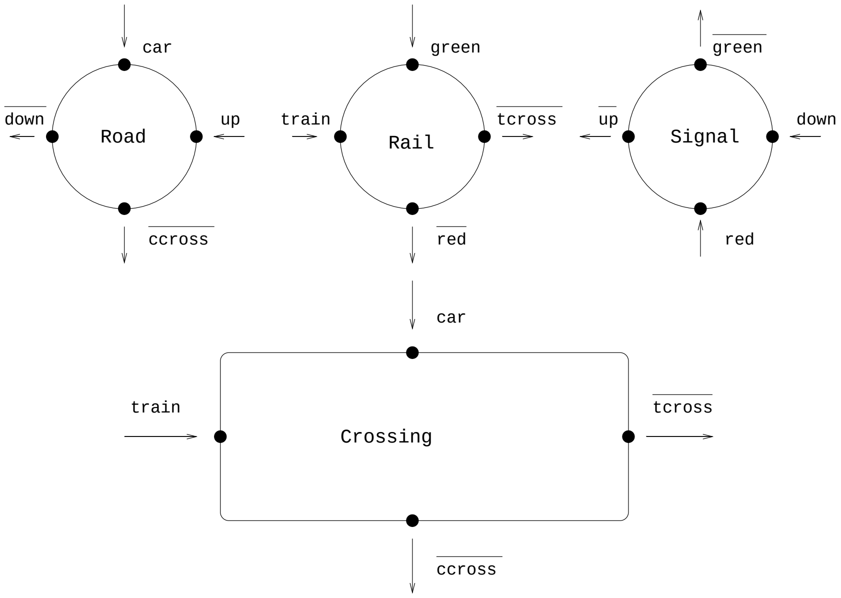
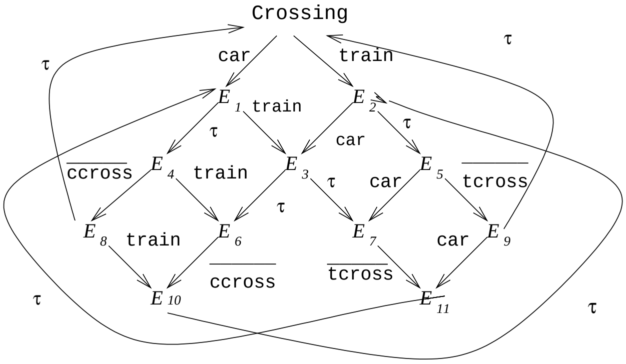


FIGURE 1.11. Flow graphs of the crossing and its components

An important arena for process descriptions is provided by modelling protocols. An example is the process `Protocol` of Figure 1.13 taken from Walker [60], which models an extremely simple communications protocol that allows a message to be lost during transmission. Its flow graph is the same as that of `Cop`, and the size of its transition graph depends on the space of messages. The sender transmits any message it receives at the port `in` to the medium. In turn, the medium may transmit the message to the receiver, or instead the message may be lost, an action modelled as the silent τ action, in which case the medium sends a timeout signal to the sender and the message is retransmitted. On receiving a message, the receiver transmits it at the port `out` and then sends an acknowledgement directly to the sender (which we assume can not be lost). Having received the acknowledgement, the sender may again receive a message at port `in`.

Although the flow graphs for `Protocol` and `Cop` are the same, their levels of detail are very different. The process `Cop` is a one-place buffer that takes in a value and later expels it. Similarly, the protocol takes in a message and later may output it. The transition graph associated with this process when there is just one message is pictured in Figure 1.14. It turns out that `Protocol` and `Cop` are observationally equivalent, as defined in Chapter 3. As process descriptions, however, they are very different. `Cop` is close to a specification, as its desired behaviour is given merely in terms of what it does. In contrast, `Protocol` is closer to an implementation, because it is defined in terms of how it is built from simpler components.

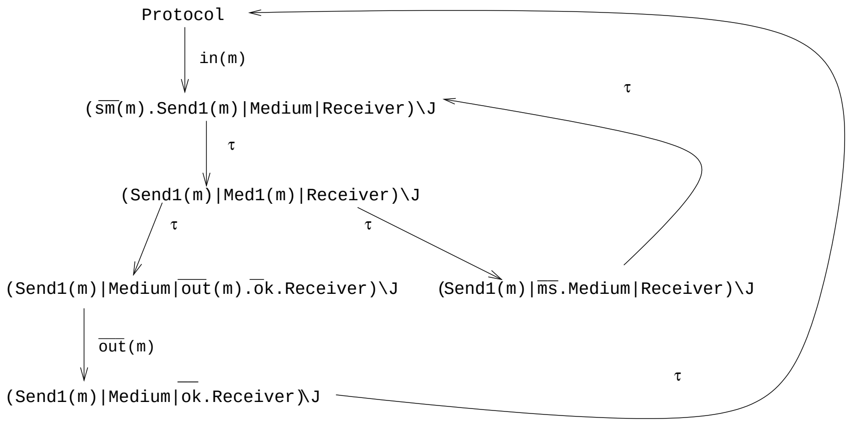


$$
\begin{array}{lcl}
K & = & \{\mathtt{green}, \mathtt{red}, \mathtt{up}, \mathtt{down}\} \\[1em]
E_1 & \equiv & (\mathtt{up}.\overline{\mathtt{ccross}}.\overline{\mathtt{down}}.\mathtt{Road} \mid \mathtt{Rail} \mid \mathtt{Signal})\backslash K \\
E_2 & \equiv & (\mathtt{Road} \mid \mathtt{green}.\overline{\mathtt{tcross}}.\overline{\mathtt{red}}.\mathtt{Rail} \mid \mathtt{Signal})\backslash K \\
E_3 & \equiv & (\mathtt{up}.\overline{\mathtt{ccross}}.\overline{\mathtt{down}}.\mathtt{Road} \mid \mathtt{green}.\overline{\mathtt{tcross}}.\overline{\mathtt{red}}.\mathtt{Rail} \mid \mathtt{Signal})\backslash K \\
E_4 & \equiv & (\overline{\mathtt{ccross}}.\overline{\mathtt{down}}.\mathtt{Road} \mid \mathtt{Rail} \mid \mathtt{down}.\mathtt{Signal})\backslash K \\
E_5 & \equiv & (\mathtt{Road} \mid \overline{\mathtt{tcross}}.\overline{\mathtt{red}}.\mathtt{Rail} \mid \mathtt{red}.\mathtt{Signal})\backslash K \\
E_6 & \equiv & (\overline{\mathtt{ccross}}.\overline{\mathtt{down}}.\mathtt{Road} \mid \mathtt{green}.\overline{\mathtt{tcross}}.\overline{\mathtt{red}}.\mathtt{Rail} \mid \mathtt{down}.\mathtt{Signal})\backslash K \\
E_7 & \equiv & (\mathtt{up}.\overline{\mathtt{ccross}}.\overline{\mathtt{down}}.\mathtt{Road} \mid \overline{\mathtt{tcross}}.\overline{\mathtt{red}}.\mathtt{Rail} \mid \mathtt{red}.\mathtt{Signal})\backslash K \\
E_8 & \equiv & (\overline{\mathtt{down}}.\mathtt{Road} \mid \mathtt{Rail} \mid \mathtt{down}.\mathtt{Signal})\backslash K \\
E_9 & \equiv & (\mathtt{Road} \mid \overline{\mathtt{red}}.\mathtt{Rail} \mid \mathtt{red}.\mathtt{Signal})\backslash K \\
E_{10} & \equiv & (\overline{\mathtt{down}}.\mathtt{Road} \mid \mathtt{green}.\overline{\mathtt{tcross}}.\overline{\mathtt{red}}.\mathtt{Rail} \mid \mathtt{down}.\mathtt{Signal})\backslash K \\
E_{11} & \equiv & (\mathtt{up}.\overline{\mathtt{ccross}}.\overline{\mathtt{down}}.\mathtt{Road} \mid \overline{\mathtt{red}}.\mathtt{Rail} \mid \mathtt{red}.\mathtt{Signal})\backslash K
\end{array}
$$

FIGURE 1.12. Transition graph of `Crossing`

$$
\begin{array}{lcl}
\mathtt{Sender} & \stackrel{\mathrm{def}}{=} & \mathtt{in}(x).\overline{\mathtt{sm}}(x).\mathtt{Send1}(x) \\
\mathtt{Send1}(x) & \stackrel{\mathrm{def}}{=} & \mathtt{ms}.\overline{\mathtt{sm}}(x).\mathtt{Send1}(x) + \mathtt{ok}.\mathtt{Sender} \\
\mathtt{Medium} & \stackrel{\mathrm{def}}{=} & \mathtt{sm}(y).\mathtt{Med1}(y) \\
\mathtt{Med1}(y) & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{mr}}(y).\mathtt{Medium} + \tau.\overline{\mathtt{ms}}.\mathtt{Medium} \\
\mathtt{Receiver} & \stackrel{\mathrm{def}}{=} & \mathtt{mr}(x).\overline{\mathtt{out}}(x).\overline{\mathtt{ok}}.\mathtt{Receiver} \\[1em]
\mathtt{Protocol} & \equiv & (\mathtt{Sender} \mid \mathtt{Medium} \mid \mathtt{Receiver})\backslash\{\mathtt{sm}, \mathtt{ms}, \mathtt{mr}, \mathtt{ok}\}
\end{array}
$$

FIGURE 1.13. A simple protocol



$$J = \{\mathtt{sm}, \mathtt{ms}, \mathtt{mr}, \mathtt{ok}\}$$

FIGURE 1.14. `Protocol` transition graph when there is one message m.

Example 2 An example of an infinite state system from Bradfield and the author [10] is the slot machine $\mathtt{SM}_n$ defined in Figure 1.15. Its flow graph is also depicted there. A coin is input (the action `slot`) and then, after some silent activity, either a loss or a winning sum of money is output. The system consists of three components: `IO`, which handles the taking and paying out of money; $\mathtt{B_n}$, a bank holding n pounds; and `D`, the wheel-spinning decision component.

Exercises

1. Give a derivation of the following transition.

$$\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}}) \xrightarrow{\tau} \overline{\mathtt{out}}(v2).\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User})$$

2. Show that the following three processes
   (a) $(\mathtt{Cop} \mid \mathtt{User_{v1}}) \mid \mathtt{User_{v2}}$
   (b) $\mathtt{User_{v1}} \mid (\mathtt{Cop} \mid \mathtt{User_{v2}})$
   (c) $\mathtt{Cop} \mid (\mathtt{User_{v1}} \mid \mathtt{User_{v2}})$

   have isomorphic transition graphs (and flow graphs).

3. $\mathtt{Sem} \overset{\mathrm{def}}{=} \mathtt{get.put.Sem}$ is a semaphore. Draw the transition graph for $\mathtt{Sem} \mid \mathtt{Sem} \mid \mathtt{Sem} \mid \mathtt{Sem}$.

4. How does the transition graph for `Cnt` differ from that for the counter $\mathtt{Ct_0}$ of Figure 1.4?

$$\begin{array}{lcl}
\mathtt{IO} & \stackrel{\mathrm{def}}{=} & \mathtt{slot}.\overline{\mathtt{bank}}.(\mathtt{lost}.\overline{\mathtt{loss}}.\mathtt{IO} + \mathtt{release}(y).\overline{\mathtt{win}}(y).\mathtt{IO}) \\
\mathtt{B_n} & \stackrel{\mathrm{def}}{=} & \mathtt{bank}.\overline{\mathtt{max}}(n+1).\mathtt{left}(y).\mathtt{B}_y \\
\mathtt{D} & \stackrel{\mathrm{def}}{=} & \mathtt{max}(z).(\overline{\mathtt{lost}}.\overline{\mathtt{left}}(z).\mathtt{D} + \sum\{\overline{\mathtt{release}}(y).\overline{\mathtt{left}}(z-y).\mathtt{D} \ : \ 1 \le y \le z\}) \\
 & & \\
\mathtt{SM_n} & \equiv & (\mathtt{IO} \mid \mathtt{B_n} \mid \mathtt{D})\backslash\{\mathtt{bank}, \mathtt{lost}, \mathtt{max}, \mathtt{left}, \mathtt{release}\}
\end{array}$$

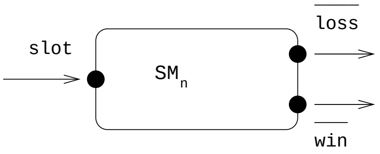


FIGURE 1.15. A slot machine

5. Draw the transition graph for $\mathtt{Bag} \stackrel{\mathrm{def}}{=} \mathtt{in}(x).(\overline{\mathtt{out}}(x).\mathtt{0} \mid \mathtt{Bag})$ when the space of values contains just two elements, 0 and 1.

6. Let L_1 be the set of actions $\{\mathtt{1p}, \mathtt{little}\}$ and let L_2 be $\{\mathtt{1p}, \mathtt{little}, \mathtt{2p}\}$. Also let $\mathtt{Use_1} \stackrel{\mathrm{def}}{=} \overline{\mathtt{1p}}.\overline{\mathtt{little}}.\mathtt{Use_1}$. Draw flow graphs and transition graphs for the processes

   (a) $\mathtt{Ven} \mid \mathtt{Use_1}$
   (b) $\mathtt{Ven} \mid (\mathtt{Use_1} \mid \mathtt{Use_1})$
   (c) $(\mathtt{Ven} \mid \mathtt{Use_1})\backslash L_i$
   (d) $(\mathtt{Ven} \mid \mathtt{Use_1})\backslash L_i \mid \mathtt{Use_1}$
   (e) $(\mathtt{Ven} \mid \mathtt{Use_1} \mid \mathtt{Use_1})\backslash L_i$

   when $i = 1$ and $i = 2$.

7. Let $\mathcal{G}(E)$ be the transition graph for E. Define prefixing (.), +, $\mid$ and $\backslash J$ operators directly on transition graphs so that each of the following pairs is isomorphic.

   (a) $a.\mathcal{G}(E)$ and $\mathcal{G}(a.E)$
   (b) $\mathcal{G}(E+F)$ and $\mathcal{G}(E) + \mathcal{G}(F)$
   (c) $\mathcal{G}(E \mid F)$ and $\mathcal{G}(E) \mid \mathcal{G}(F)$
   (d) $\mathcal{G}(E)\backslash J$ and $\mathcal{G}(E\backslash J)$

8. Consider the definition of the following process from Hennessy and Ingolfsdottir [27].

$$\begin{array}{lcl}
\mathtt{Fac} & \stackrel{\mathrm{def}}{=} & \mathtt{in_1}(y).\mathtt{in_2}(z).\textbf{if}\ y = 0\ \textbf{then}\ \overline{\mathtt{out}}(z).\mathtt{0} \\
 & & \qquad\qquad\qquad\quad \textbf{else}\ (\overline{\mathtt{in_1}}(y-1).\overline{\mathtt{in_2}}(z*y).\mathtt{0} \mid \mathtt{Fac})
\end{array}$$

   Draw the transition graph of $(\overline{\mathtt{in_1}}(3).\overline{\mathtt{in_2}}(1).\mathtt{0} \mid \mathtt{Fac})\backslash\{\mathtt{in_1}, \mathtt{in_2}\}$.

9. Draw the transition graph for `Road` | `Rail` | `Signal`, and compare it with that for `Crossing`.
10. Draw flow and transition graphs for the components of `Protocol`.
11. Refine the description of `Protocol` so that acknowledgements may also be lost.

1.3 Observable transitions

Actions a on the transition relations $\xrightarrow{a}$ between processes can be extended to finite length sequences w, which are also called "traces." The extended transition $E \xrightarrow{w} F$ states that E may perform the trace w and become F. There are two transition rules for traces, where ε is the empty sequence of actions.

$$\boxed{\mathrm{R(tr)}\quad E \xrightarrow{\varepsilon} E \qquad \dfrac{E \xrightarrow{aw} F}{E \xrightarrow{a} E' \quad E' \xrightarrow{w} F}}$$

First is the axiom that any process may carry out the empty sequence and remain unchanged. The second rule allows traces to be extended. If $E \xrightarrow{a} E'$ and E' can perform the trace w and become F then $E \xrightarrow{aw} F$. No distinction is made between carrying out the action a and carrying out the trace a (understood as an action sequence of length one). Below is the derivation of the extended transition $\mathtt{Ven_b} \xrightarrow{\mathtt{big}\,\mathtt{collect_b}} \mathtt{Ven}$ when $\mathtt{Ven_b}$ is part of the vending machine of Section 1.1.

$$\dfrac{\mathtt{Ven_b} \xrightarrow{\mathtt{big}\,\mathtt{collect_b}} \mathtt{Ven}}{\dfrac{\mathtt{Ven_b} \xrightarrow{\mathtt{big}} \mathtt{collect_b.Ven}}{\mathtt{big.collect_b.Ven} \xrightarrow{\mathtt{big}} \mathtt{collect_b.Ven}} \qquad \mathtt{collect_b.Ven} \xrightarrow{\mathtt{collect_b}} \mathtt{Ven}}$$

Internal τ actions have a different status from incomplete actions. An incomplete action is "observable" because it is susceptible of interaction in a parallel context. Suppose that E may at some time perform the action `ok`, and that `Resource` is a resource. In the context $(E \mid \overline{\mathtt{ok}}.\mathtt{Resource})\backslash\{\mathtt{ok}\}$ access to `Resource` is only triggered with an execution of `ok` by E. Observation of `ok` is the same as the release of `Resource`. The silent action τ cannot be observed in this way. Consequently, an important abstraction of process behaviour derives from silent activity.

Consider the following copier `C` and the user `U`.

$$\begin{aligned} \mathtt{C} &\overset{\mathrm{def}}{=} \mathtt{in}(x).\overline{\mathtt{out}}(x).\overline{\mathtt{ok}}.\mathtt{C} \\ \mathtt{U} &\overset{\mathrm{def}}{=} \mathtt{write}(x).\overline{\mathtt{in}}(x).\mathtt{ok}.\mathtt{U} \end{aligned}$$

$\mathtt{U}$ writes a file before sending it through $\mathtt{in}$ and then waits for an acknowledgement. $(\mathtt{C} \mid \mathtt{U})\backslash\{\mathtt{in}, \mathtt{ok}\}$ has similar behaviour to $\mathtt{Ucop}$.

$$\mathtt{Ucop} \overset{\text{def}}{=} \mathtt{write}(x).\overline{\mathtt{out}}(x).\mathtt{Ucop}$$

The only difference in their abilities is internal activity. Both are initially able only to carry out a write action

$$\begin{array}{l} \mathtt{Ucop} \xrightarrow{\mathtt{write}(v)} \overline{\mathtt{out}}(v).\mathtt{Ucop} \\ (\mathtt{C} \mid \mathtt{U})\backslash\{\mathtt{in}, \mathtt{ok}\} \xrightarrow{\mathtt{write}(v)} (\mathtt{C} \mid \overline{\mathtt{in}}(v).\mathtt{ok}.\mathtt{U})\backslash\{\mathtt{in}, \mathtt{ok}\}. \end{array}$$

Process $\overline{\mathtt{out}}(v).\mathtt{Ucop}$ outputs immediately, whereas the other process must first perform a communication before it outputs, and then τ again before a second write can happen. By abstracting from silent behaviour, this difference disappears. Outwardly, both processes repeatedly write and output.

A trace w is a sequence of actions. The trace $w \upharpoonright J$ is the subsequence of w when actions that do not belong to J are erased.

$$\begin{array}{rcl} \varepsilon \upharpoonright J & = & \varepsilon \\ aw \upharpoonright J & = & \begin{cases} a(w \upharpoonright J) & \text{if } a \in J \\ w \upharpoonright J & \text{otherwise} \end{cases} \end{array}$$

Below are three simple examples.

$$\begin{array}{lcl} (\mathtt{train}\,\tau\,\overline{\mathtt{tcross}}\,\tau) \upharpoonright \{\overline{\mathtt{tcross}}\} & = & \overline{\mathtt{tcross}} \\ (\tau\,\overline{\mathtt{ccross}}\,\tau) \upharpoonright \{\overline{\mathtt{tcross}}\} & = & \varepsilon \\ (\mathtt{write}(v)\,\tau\,\overline{\mathtt{out}}(v)\,\tau) \upharpoonright \{\mathtt{write}, \overline{\mathtt{out}}\} & = & \mathtt{write}(v)\,\overline{\mathtt{out}}(v) \end{array}$$

Associated with any trace w is the *observable* trace $w \upharpoonright \mathsf{O}$, where O is a universal set of observable actions containing at least all actions mentioned in this work apart from τ. The effect of $\upharpoonright \mathsf{O}$ on w is to erase all occurrences of the silent action τ, as illustrated by the following examples.

$$\begin{array}{lcl} (\mathtt{in}(m)\,\tau\,\tau\,\overline{\mathtt{out}}(m)\,\tau) \upharpoonright \mathsf{O} & = & \mathtt{in}(m)\,\overline{\mathtt{out}}(m) \\ (\mathtt{in}(m)\,\tau\,\tau\,\tau\,\tau) \upharpoonright \mathsf{O} & = & \mathtt{in}(m) \\ (\tau\,\tau\,\tau\,\tau\,\tau\,\tau) \upharpoonright \mathsf{O} & = & \varepsilon \end{array}$$

To capture observable behaviour, another family of transition relations between processes is introduced. $E \overset{u}{\Longrightarrow} F$ expresses that E may carry out the observable trace u and become F. The transition rule for observable traces is as follows.

$$\boxed{\mathrm{R(Tr)} \quad \frac{E \overset{u}{\Longrightarrow} F}{E \xrightarrow{w} F} \quad u = w \upharpoonright \mathsf{O}}$$

An example is $\mathtt{Protocol} \overset{\mathtt{in}(m)\,\overline{\mathtt{out}}(m)}{\Longrightarrow} \mathtt{Protocol}$, whose derivation utilises the extended transition $\mathtt{Protocol} \xrightarrow{\mathtt{in}(m)\,\tau\,\tau\,\overline{\mathtt{out}}(m)\,\tau} \mathtt{Protocol}$.

Observable traces can also be built from their component observable actions. The extended transition $\texttt{Crossing} \overset{\texttt{train}\,\overline{\texttt{tcross}}}{\Longrightarrow} \texttt{Crossing}$ is the result of gluing together $\texttt{Crossing} \overset{\texttt{train}}{\Longrightarrow} E$ and $E \overset{\overline{\texttt{tcross}}}{\Longrightarrow} \texttt{Crossing}$ when the intermediate state E is E_2 or E_5 of Figure 1.12. Observable behaviour is constructed from transitions $E \overset{\varepsilon}{\Longrightarrow} F$ or $E \overset{a}{\Longrightarrow} F$ when $a \in \mathsf{O}$, whose rules are as follows.

$$\boxed{\mathrm{R}(\overset{\varepsilon}{\Longrightarrow}) \quad E \overset{\varepsilon}{\Longrightarrow} E \qquad \dfrac{E \overset{\varepsilon}{\Longrightarrow} F}{E \xrightarrow{\tau} E' \quad E' \overset{\varepsilon}{\Longrightarrow} F}}$$

$$\boxed{\mathrm{R}(\overset{a}{\Longrightarrow}) \qquad \dfrac{E \overset{a}{\Longrightarrow} F}{E \overset{\varepsilon}{\Longrightarrow} E' \quad E' \xrightarrow{a} F' \quad F' \overset{\varepsilon}{\Longrightarrow} F}}$$

$E \overset{\varepsilon}{\Longrightarrow} F$ if E can silently evolve to F and $E \overset{a}{\Longrightarrow} F$ if E can silently evolve to a process that carries out a and then silently becomes F.

Example 1 The derivation of $\texttt{Protocol} \overset{\texttt{in}(m)}{\Longrightarrow} F_3$, where F_3 abbreviates $(\texttt{Send1}(m) \mid \texttt{Medium} \mid \overline{\texttt{out}}(m).\overline{\texttt{ok}}.\texttt{Receiver})\backslash\{\texttt{sm}, \texttt{ms}, \texttt{mr}, \texttt{ok}\}$, uses the following two intermediate states (see Figure 1.14).

$$\begin{aligned} F_1 &\equiv (\overline{\texttt{sm}}(m).\texttt{Send1}(m) \mid \texttt{Medium} \mid \texttt{Receiver})\backslash\{\texttt{sm}, \texttt{ms}, \texttt{mr}, \texttt{ok}\} \\ F_2 &\equiv (\texttt{Send1}(m) \mid \texttt{Med1}(m) \mid \texttt{Receiver})\backslash\{\texttt{sm}, \texttt{ms}, \texttt{mr}, \texttt{ok}\} \end{aligned}$$

Below is part of the derivation.

$$\dfrac{\texttt{Protocol} \overset{\texttt{in}(m)}{\Longrightarrow} F_3}{\texttt{Protocol} \overset{\varepsilon}{\Longrightarrow} \texttt{Protocol} \quad \texttt{Protocol} \xrightarrow{\texttt{in}(m)} F_1 \quad F_1 \overset{\varepsilon}{\Longrightarrow} F_3}$$

Part of the derivation of $F_1 \overset{\varepsilon}{\Longrightarrow} F_3$ is as follows.

$$\dfrac{F_1 \overset{\varepsilon}{\Longrightarrow} F_3}{F_1 \xrightarrow{\tau} F_2 \qquad \dfrac{F_2 \overset{\varepsilon}{\Longrightarrow} F_3}{F_2 \xrightarrow{\tau} F_3 \quad F_3 \overset{\varepsilon}{\Longrightarrow} F_3}}$$

Observable behaviour of a process can also be visually encapsulated as a transition graph. As in Section 1.1, ingredients of this graph are process terms related by transitions. Each edge has the form $\overset{\varepsilon}{\Longrightarrow}$ or $\overset{a}{\Longrightarrow}$ when $a \in \mathsf{O}$. Assuming a value space with just one element v, the observable transition graphs for $(\texttt{C} \mid \texttt{U})\backslash\{\texttt{in}, \texttt{ok}\}$ and $\texttt{Ucop}$ are pictured in Figure 1.16 (where thick arrows are used instead of $\Longrightarrow$).

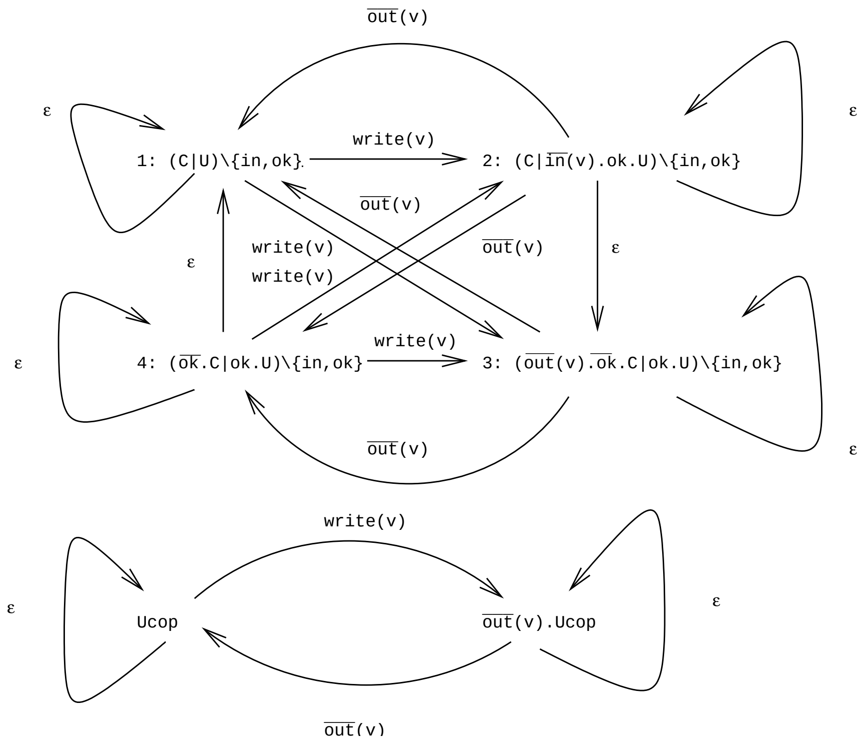


FIGURE 1.16. Observable transition graphs for $(\texttt{C} \mid \texttt{U})\backslash\{\texttt{in},\texttt{ok}\}$ and `Ucop`

There are *two* behaviour graphs associated with any process. Although both graphs contain the same vertices, they differ in their labelled edges. Observable graphs are more complex, since they contain more transitions. However, this abundance of transitions may result in redundant vertices. Figure 1.16 exemplifies this condition in the case of $(\mathtt{C} \mid \mathtt{U})\backslash\{\mathtt{in}, \mathtt{ok}\}$. The states labelled 1 and 4 have identical capabilities, as do the states labelled 2 and 3. When minimized with respect to observable equivalences, as defined in Chapter 3, these graphs may be dramatically simplified as their vertices are fused.

Exercises

1. Derive the extended transition $\mathtt{SM_n} \xrightarrow{w} \mathtt{SM_{n+1}}$ when w is the following trace $\mathtt{slot}\,\tau\,\tau\,\tau\,\tau\,\overline{\mathtt{loss}}$ and $\mathtt{SM_n}$ is the slot machine.
2. Provide a full derivation of $\mathtt{Protocol} \xrightarrow{s} \mathtt{Protocol}$ when s is the trace $\mathtt{in}(m)\,\tau\,\tau\,\overline{\mathtt{out}}(m)\,\tau$.
3. List the members of the following sets:

$$\{E \,:\, \mathtt{Crossing} \overset{\mathtt{train}\,\overline{\mathtt{tcross}}}{\Longrightarrow} E\}$$
$$\{E \,:\, \mathtt{Protocol} \overset{\mathtt{in}(m)}{\Longrightarrow} E\}$$

4. Show that $E \overset{a}{\Longrightarrow} F$ is derivable via the rules R(tr) and R(Tr) iff it is derivable using the rules R($\overset{a}{\Longrightarrow}$) and R($\overset{\varepsilon}{\Longrightarrow}$).
5. Draw the observable transition graphs for the processes: `Cl`, `Ven` and `Crossing`.
6. Although observable traces abstract from silent activity, this does not mean that internal actions can not contribute to differences in observable capability. Let $\mathtt{Ven}'$ be a vending machine very similar to `Ven` of Figure 1.2, except that the initial `2p` action is prefaced by the silent action, $\mathtt{Ven}' \overset{\text{def}}{=} \tau.\mathtt{2p}.\mathtt{Ven_b} + \mathtt{1p}.\mathtt{Ven_l}$
   - (a) Show that `Ven` and $\mathtt{Ven}'$ have the same observable traces.
   - (b) Let $\mathtt{Use_1}$ be the user $\mathtt{Use_1} \overset{\text{def}}{=} \overline{\mathtt{1p}}.\overline{\mathtt{little}}.\mathtt{Use_1}$, who is only interested in inserting the smaller coin. Show that the process $(\mathtt{Ven}' \mid \mathtt{Use_1})\backslash\{\mathtt{1p}, \mathtt{2p}, \mathtt{little}\}$ may deadlock before an observable action is carried out unlike $(\mathtt{Ven} \mid \mathtt{Use_1})\backslash\{\mathtt{1p}, \mathtt{2p}, \mathtt{little}\}$.
   - (c) Draw both kinds of transition graphs for each of the processes in part (b).
7. Assuming just one datum value, draw the observable graphs for processes $(\mathtt{Cop} \mid \mathtt{User})\backslash\{\mathtt{in}\}$ and `Protocol`. What states of these graphs can be fused together?

8. Let $\mathcal{G}(E)$ be the transition graph for E, and let $\mathcal{G}^o(E)$ be its observable transition graph. Define the graph transformation ^{o} that maps $\mathcal{G}(E)$ into $\mathcal{G}^o(E)$.
9. A process is said to be "divergent" if it can perform the τ action forever.
   - (a) Draw both kinds of transition graph for the following pair of processes, $\tau.\mathtt{0}$ and $\mathtt{Div}' \overset{\mathrm{def}}{=} \tau.\mathtt{Div}' + \tau.\mathtt{0}$.
   - (b) Do you think that the processes `Protocol` and `Cop` have the same observable behaviour? Give reasons for and against.

1.4 Renaming and linking

`Cop`, `User` and `Ucop` of previous sections are essentially one-place buffers, taking in a value and later expelling it. Assume that `B` is the following canonical buffer.

$$\mathtt{B} \overset{\mathrm{def}}{=} \mathtt{i}(x).\overline{\mathtt{o}}(x).\mathtt{B}$$

For instance, `Cop` is the process `B` when port `i` is `in` and port `o` is `out`. Relabelling of ports can be made explicit by introducing an operator which renames actions.

The crux of renaming is a function mapping actions into actions. To ensure pleasant properties, a renaming function f is subject to a few restrictions. First, it should respect complements. For any observable a, the actions $f(a)$ and $f(\overline{a})$ are co-actions, that is $f(\overline{a}) = \overline{f(a)}$. Second, it should conserve the silent action, $f(\tau) = \tau$. Associated with any function f obeying these conditions is the renaming operator $[f]$, which, when applied to process E, is written as $E[f]$; this is the process E whose actions are relabelled according to f.

A renaming function f can be abbreviated to its essential part. If each a_i is a distinct observable action, then $b_1/a_1, \ldots, b_n/a_n$ represents the function f that renames a_i to b_i (and $\overline{a_i}$ to $\overline{b_i}$), and leaves any other action c unchanged. For instance, `Cop` abbreviates the process $\mathtt{B}[\mathtt{in}/\mathtt{i}, \mathtt{out}/\mathtt{o}]$: here we maintain the convention that `in` stands for the family $\{\mathtt{in}(v) : v \in D\}$ and `i` for $\{\mathtt{i}(v) : v \in D\}$, so $\mathtt{in}/\mathtt{i}$ symbolises the function that also preserves values by mapping $\mathtt{i}(v)$ to $\mathtt{in}(v)$ for each v. The transition rule for renaming is set forth below.

$$\boxed{\mathrm{R}([f]) \quad \frac{E[f] \xrightarrow{a} F[f]}{E \xrightarrow{b} F} \quad a = f(b)}$$

This rule is used in derivations of the following pair of transitions.

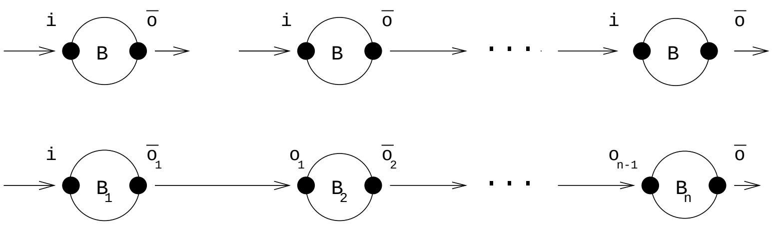


FIGURE 1.17. Flow graph of n instances of $\mathtt{B}$, and $\mathtt{B}_1 \mid \ldots \mid \mathtt{B}_n$.

$$\mathtt{B}[\mathtt{in}/\mathtt{i}, \mathtt{out}/\mathtt{o}] \xrightarrow{\mathtt{in}(v)} (\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{in}/\mathtt{i}, \mathtt{out}/\mathtt{o}] \xrightarrow{\overline{\mathtt{out}}(v)} \mathtt{B}[\mathtt{in}/\mathtt{i}, \mathtt{out}/\mathtt{o}]$$

Below is the derivation of the initial transition.

$$\begin{array}{c} \mathtt{B}[\mathtt{in}/\mathtt{i}, \mathtt{out}/\mathtt{o}] \xrightarrow{\mathtt{in}(v)} (\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{in}/\mathtt{i}, \mathtt{out}/\mathtt{o}] \\ \hline \mathtt{B} \xrightarrow{\mathtt{i}(v)} \overline{\mathtt{o}}(v).\mathtt{B} \\ \hline \mathtt{i}(x).\overline{\mathtt{o}}(x).\mathtt{B} \xrightarrow{\mathtt{i}(v)} \overline{\mathtt{o}}(v).\mathtt{B} \end{array}$$

A virtue of process modelling is that it allows building systems from simpler components. Consider how to model an n-place buffer when $n > 1$, following Milner [44], by linking together n instances of $\mathtt{B}$ in parallel. The flow graph of n copies of $\mathtt{B}$ is pictured in Figure 1.17. For this to become an n-place buffer we need to "link," and then internalise, the contiguous $\overline{\mathtt{o}}$ and $\mathtt{i}$ ports. Renaming permits linking, as the following variants of $\mathtt{B}$ show.

$$\begin{array}{lll} \mathtt{B}_1 & \equiv & \mathtt{B}[\mathtt{o}_1/\mathtt{o}] \\ \mathtt{B}_{j+1} & \equiv & \mathtt{B}[\mathtt{o}_j/\mathtt{i}, \mathtt{o}_{j+1}/\mathtt{o}] \quad 1 \leq j < n-1 \\ \mathtt{B}_n & \equiv & \mathtt{B}[\mathtt{o}_{n-1}/\mathtt{i}] \end{array}$$

The flow graph of $\mathtt{B}_1 \mid \ldots \mid \mathtt{B}_n$ is also shown in Figure 1.17, and contains the intended links. The n-place buffer is the result of internalizing these contiguous links, $(\mathtt{B}_1 \mid \ldots \mid \mathtt{B}_n)\backslash\{\mathtt{o}_1, \ldots, \mathtt{o}_{n-1}\}$.

Part of the behaviour of a two-place buffer is illustrated by the following cycle.

$$\begin{tikzcd}
(\mathtt{B}[\mathtt{o}_1/\mathtt{o}] \mid \mathtt{B}[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\} \arrow[r, "\mathtt{i}(v)"] & ((\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{o}] \mid \mathtt{B}[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\} \arrow[d, "\tau"] \\
((\overline{\mathtt{o}}(w).\mathtt{B})[\mathtt{o}_1/\mathtt{o}] \mid (\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\} \arrow[d, "\overline{\mathtt{o}}(v)"] & (\mathtt{B}[\mathtt{o}_1/\mathtt{o}] \mid (\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\} \arrow[l, "\mathtt{i}(w)"'] \\
((\overline{\mathtt{o}}(w).\mathtt{B})[\mathtt{o}_1/\mathtt{o}] \mid \mathtt{B}[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\} \arrow[r, "\tau"] & (\mathtt{B}[\mathtt{o}_1/\mathtt{o}] \mid (\overline{\mathtt{o}}(w).\mathtt{B})[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\} \arrow[d, "\overline{\mathtt{o}}(w)"] \\
 & (\mathtt{B}[\mathtt{o}_1/\mathtt{o}] \mid \mathtt{B}[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\}
\end{tikzcd}$$

Below is the derivation of the second transition.

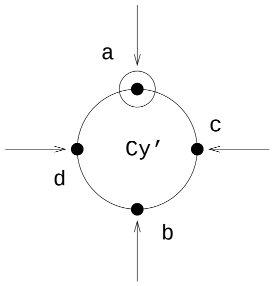


FIGURE 1.18. The flow graph of $\mathtt{Cy}'$

$$\dfrac{((\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{o}] \mid \mathtt{B}[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\} \xrightarrow{\tau} (\mathtt{B}[\mathtt{o}_1/\mathtt{o}] \mid (\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{i}])\backslash\{\mathtt{o}_1\}}{\dfrac{(\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{o}] \mid \mathtt{B}[\mathtt{o}_1/\mathtt{i}] \xrightarrow{\tau} \mathtt{B}[\mathtt{o}_1/\mathtt{o}] \mid (\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{i}]}{\dfrac{(\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{o}] \xrightarrow{\overline{\mathtt{o}}_1(v)} \mathtt{B}[\mathtt{o}_1/\mathtt{o}]}{\overline{\mathtt{o}}(v).\mathtt{B} \xrightarrow{\overline{\mathtt{o}}(v)} \mathtt{B}} \quad \dfrac{\mathtt{B}[\mathtt{o}_1/\mathtt{i}] \xrightarrow{\mathtt{o}_1(v)} (\overline{\mathtt{o}}(v).\mathtt{B})[\mathtt{o}_1/\mathtt{i}]}{\dfrac{\mathtt{B} \xrightarrow{\mathtt{i}(v)} \overline{\mathtt{o}}(v).\mathtt{B}}{\mathtt{i}(x).\overline{\mathtt{o}}(x).\mathtt{B} \xrightarrow{\mathtt{i}(v)} \overline{\mathtt{o}}(v).\mathtt{B}}}}}$$

A more involved example from Milner [44] refers to the construction of a scheduler from small cycling components. Assume n tasks when $n > 1$, and that action a_i initiates the ith task, whereas b_i signals its completion. The scheduler plans the order of task initiation, ensuring that the sequence of actions $a_1 \ldots a_n$ is carried out cyclically starting with a_1. The tasks may terminate in any order, but a task can not be restarted until its previous operation has finished. So, the scheduler must guarantee that the actions a_i and b_i happen alternately for each i.

Let $\mathtt{Cy}'$ be a cycler of length four, $\mathtt{Cy}' \overset{\mathrm{def}}{=} a.c.b.d.\mathtt{Cy}'$, whose flow graph is illustrated in Figure 1.18. In this case, the flow graph is very close to its transition graph, so we have circled the a label to indicate that it is initially active. As soon as a happens, control passes to the active action c. The clockwise movement of activity around this flowgraph is its transition graph. A first attempt at building the required scheduler is as a ring of n cyclers, where the a action is task initiation, the b action is task termination, and the other actions c and d are used for synchronization.

$$\begin{array}{lcl} \mathtt{Cy}'_1 & \equiv & \mathtt{Cy}'[a_1/a, c_1/c, b_1/b, \overline{c}_n/d] \\ \mathtt{Cy}'_i & \equiv & (d.\mathtt{Cy}')[a_i/a, c_i/c, b_i/b, \overline{c}_{i-1}/d] \quad 1 < i \leq n \end{array}$$

$\mathtt{Cy}'_1$ carries out the cycle $\mathtt{Cy}'_1 \xrightarrow{a_1\, c_1\, b_1\, \overline{c}_n} \mathtt{Cy}'_1$ and $\mathtt{Cy}'_i$, for $i > 1$ carries out the different cycle $\mathtt{Cy}'_i \xrightarrow{\overline{c}_{i-1}\, a_i\, c_i\, b_i} \mathtt{Cy}'_i$.

The flow graph of the process $\mathtt{Cy}'_1 \mid \mathtt{Cy}'_2 \mid \mathtt{Cy}'_3 \mid \mathtt{Cy}'_4$ with initial active transitions marked is pictured in Figure 1.19. Next, the c_i actions are inter-

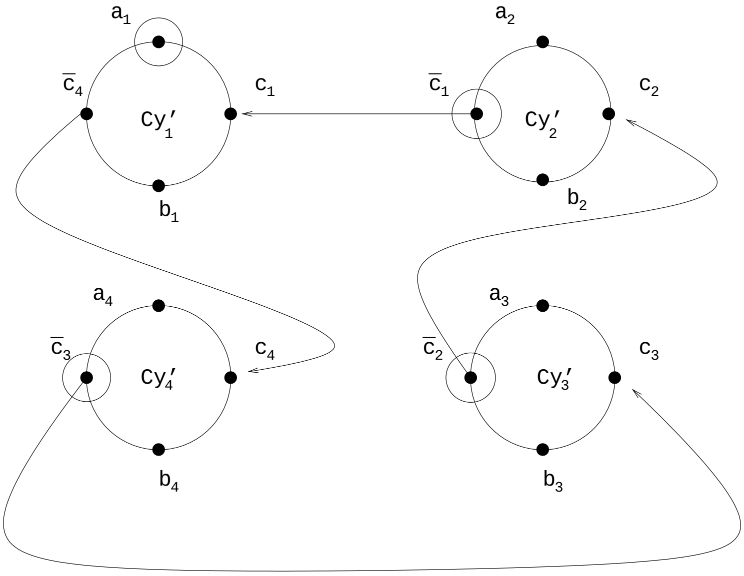


FIGURE 1.19. Flow graph of $\mathtt{Cy}_1' \mid \mathtt{Cy}_2' \mid \mathtt{Cy}_3' \mid \mathtt{Cy}_4'$

nalised. Assume that $\mathtt{Sched}_4' \equiv (\mathtt{Cy}_1' \mid \mathtt{Cy}_2' \mid \mathtt{Cy}_3' \mid \mathtt{Cy}_4')\backslash\{c_1, \ldots, c_4\}$. Imagine that the c_i actions are concealed in Figure 1.19, and notice then how the tasks must be initiated cyclically. For example, a_3 can only happen once a_1, and then a_2, have both happened. Moreover, no task can be reinitiated until its previous execution has terminated. For example, a_3 can not recur until b_3 has happened. However, $\mathtt{Sched}_4'$ does not permit all possible acceptable behaviour. Put simply, action b_4 cannot happen before b_1 because of the synchronization between c_4 and $\overline{c}_4$, meaning task four cannot terminate before the initial task.

Milner's solution in [44] to this problem is to redefine the cycler

$$\mathtt{Cy} \stackrel{\text{def}}{=} a.c.(b.d.\mathtt{Cy} + d.b.\mathtt{Cy})$$

and to use the same renaming functions. Let $\mathtt{Cy}_i$ for $1 < i \leq n$ be the process

$$(d.\mathtt{Cy})[a_i/a, c_i/c, b_i/b, \overline{c}_{i-1}/d]$$

and let $\mathtt{Cy}_1$ be $\mathtt{Cy}[a_1/a, c_1/c, b_1/b, \overline{c}_n/d]$. The required scheduler is $\mathtt{Sched}_n$, the process $(\mathtt{Cy}_1 \mid \ldots \mid \mathtt{Cy}_n)\backslash\{c_1, \ldots, c_n\}$.

Exercises

1. Redefine `Road` and `Rail` from Section 1.2 as abbreviations of $\mathtt{Cy}'$ plus renaming.

2. Assuming that the space of values consists of one element, draw both kinds of transition graph for the three-place buffer

$$(\mathtt{B}_1 \mid \mathtt{B}_2 \mid \mathtt{B}_3)\backslash\{\mathtt{o}_1, \mathtt{o}_2\}.$$

3. What extra condition on a renaming function f is necessary to ensure that the transition graphs of $(E \mid F)[f]$ and $E[f] \mid F[f]$ be isomorphic? Do either of the buffer and scheduler examples fulfil this condition?

4. (a) Draw both kinds of transition graph for the processes $\mathtt{Sched}_4$ and $\mathtt{Sched}'_4$.
   (b) Prove that $\mathtt{Sched}'_4$ permits all, and only the acceptable, behaviour of a scheduler (as described earlier).

5. From Milner [44]. Construct a sorting machine from simple components for each $n \geq 1$ capable of sorting n-length sequences of natural numbers greater than 0. It accepts exactly n numbers, one by one at `in`, then delivers them up one by one in descending order at $\overline{\mathtt{out}}$, terminated by a 0. Thereafter, it returns to its initial state.

1.5 More combinations of processes

In previous sections we have emphasised the process combinators of CCS. There is a variety of process calculi dedicated to precise modelling of systems. Besides CCS and CSP, there is ACP, due to Bergstra and Klop [5, 3], Hennessy's EPL [26], MEIJE defined by Austry, Boudol and Simone [2, 51], Milner's SCCS [43], and Winskel's general process algebra [62]. Although the behavioural meaning of all the operators of these calculi can be presented using inference rules, their conception reflects different concerns. ACP is primarily algebraic, highlighting equations[5]. CSP was devised with a distinguished model in mind, the failures model[6], and MEIJE was introduced as a very expressive calculus, initiating general results about families of transition rules that can be used to define process operators; see Groote and Vaandrager [25]. The general process algebra in [62] has roots in category theory. Moreover, users of process notation can introduce their own operators according to the application at hand.

Numerous parallel operators are proposed within the calculi mentioned above. Their transition rules are of two kinds. First, where × is parallel, is a synchronization rule.

---

[5] See Section 3.6.

[6] See Section 2.2 for the notion of failure.