Theory of concurrency

Lecture 7

Concurrency

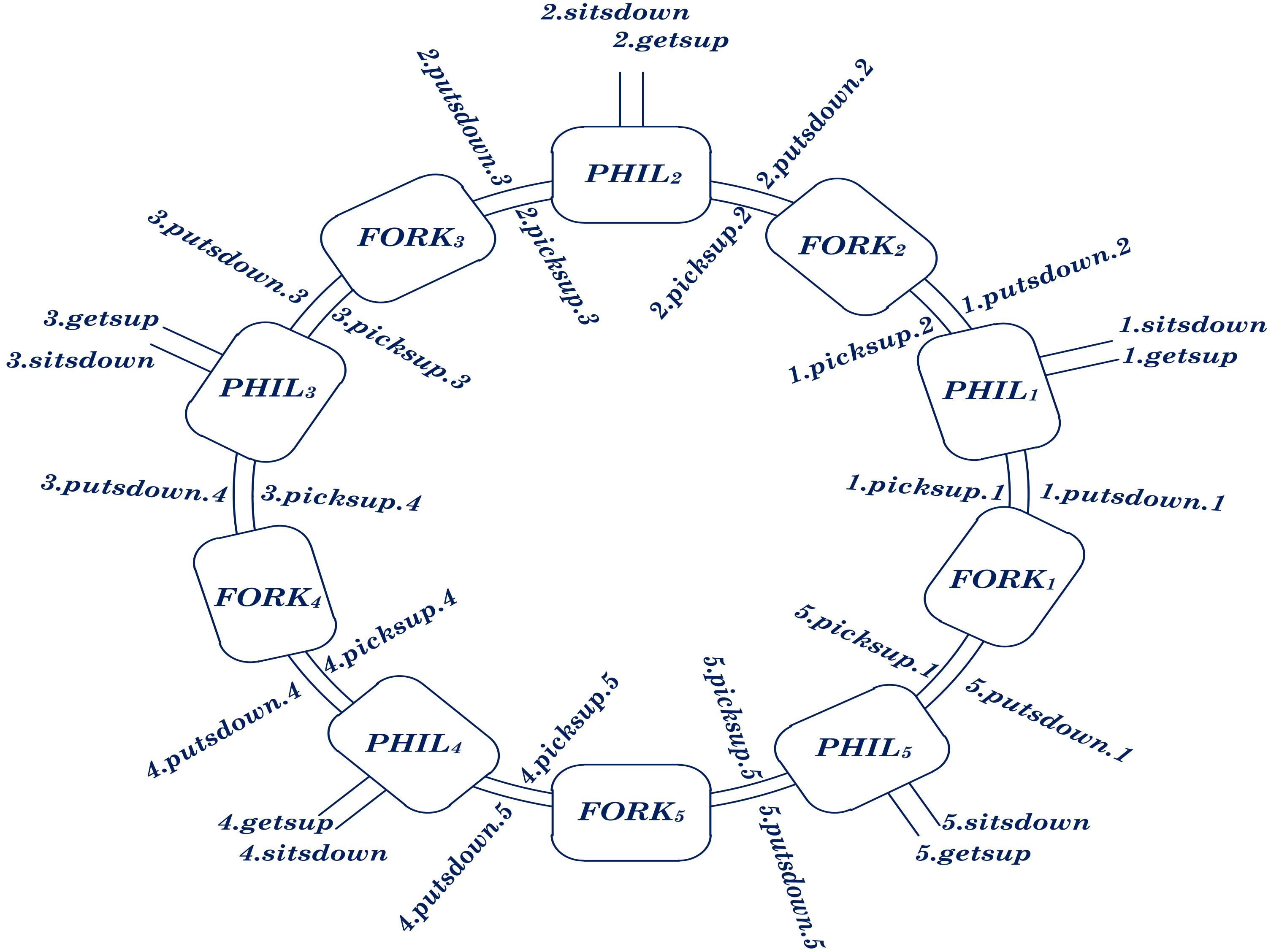
Concurrency: **Example: The Dining Philosophers**

- A College for accommodation of *five eminent philosophers*.
  - Each philosopher has a room for his professional activity of thinking
  - there is also a dining room with a circular table, surrounded by five chairs labelled by the names of the philosophers in anticlockwise manner.
    - $PHIL_0$, $PHIL_1$, $PHIL_2$, $PHIL_3$, $PHIL_4$,
    - To the left of each philosopher there is laid a golden fork
    - In the centre stays a large bowl of spaghetti replenished constantly .
  - A philosopher is expected to spend most of his time thinking
  - When he feel hungry, he goes to the dining room, sit down in his own chair,
    - picked up his own fork on his left and plunged it into the spaghetti.
      - a second fork is required to carry it to the mouth, and he picks it up on his right.
  - When he finishes, he put down his forks, get up from his chair, and continue thinking.
  - A fork can be used by only one philosopher at a time.
    - If the other philosopher wants it, he has to wait until the fork is available again.

Concurrency: The Dining Philosophers: **Alphabets**

- A mathematical model of this system.
  - The relevant sets of events.
- $aPHIL_i$ = {*i.sits down, i.gets up, i.picks up fork.i, i.picks up fork.*($i \oplus 1$)*,*
  i.puts down fork.i, i.puts down fork.($i \oplus 1$) }
  - $\oplus$ is addition modulo *5*.
- The alphabets of the philosophers are mutually disjoint.
  - no event in which they participate jointly
    - no way in which they can interact or communicate with each other.
- $aFORK_i$ = {*i.picks up fork.i,* ($i \ominus 1$)*.picks up fork.i,*
  i.puts down fork.i, ($i \ominus 1$)*.puts down fork.i*}
  - $\ominus$ is subtraction modulo *5*.
- Each event except sitting down and getting up requires
  - participation of *exactly two adjacent actors*, a philosopher and a fork

Concurrency: The Dining Philosophers: **Alphabets**



Concurrency: The Dining Philosophers: **Behaviour**

- The life of each philosopher is described as the repetition of a cycle of six events

$$
\begin{aligned}
PHIL_i = (&i.sits\ down \rightarrow \\
&\quad i.picks\ up\ fork.i \rightarrow \\
&\quad\quad i.picks\ up\ fork.(i \oplus 1) \rightarrow \\
&\quad\quad\quad i.puts\ down\ fork.i \rightarrow \\
&\quad\quad\quad\quad i.puts\ down\ fork.(i \oplus 1) \rightarrow \\
&\quad\quad\quad\quad\quad i.gets\ up \rightarrow PHIL_i)
\end{aligned}
$$

- Every fork is repeatedly picked up and put down by one of its adjacent philosophers:

$$
\begin{aligned}
FORK_i = (&i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow FORK_i \\
|\ &(i \ominus 1).picks\ up\ fork.i \rightarrow (i \ominus 1).puts\ down\ fork.i \rightarrow FORK_i)
\end{aligned}
$$

Concurrency: The Dining Philosophers: **Behaviour**

- The behaviour of the whole College is the concurrent combination of the behaviour of each of these components:

$$PHILOS = (PHIL_0 \parallel PHIL_1 \parallel PHIL_2 \parallel PHIL_3 \parallel PHIL_4)$$

$$FORKS = (FORK_0 \parallel FORK_1 \parallel FORK_2 \parallel FORK_3 \parallel FORK_4)$$

$$COLLEGE = PHILOS \parallel FORKS$$

Concurrency: The Dining Philosophers: **Behaviour**

- A variation: the philosophers can
  - pick up their two forks in either order, or put them down in either order.
- The behaviour of each philosopher's hand:
- $aLEFT_i = \{i.picks\ up\ fork.i,\ i.puts\ down\ fork.i,\ i.sits\ down,\ i.gets\ up\}$
- $aRIGHT_i = \{i.picks\ up\ fork.(i \oplus 1),\ i.puts\ down\ fork.(i \oplus 1),\ i.sits\ down,\ i.gets\ up\}$
- Each hand is capable of picking up the relevant fork, but
  - both hands are needed for sitting down and getting up
    - no fork can be raised except when the relevant philosopher is seated.
    - Apart from this, operations on the two forks are arbitrarily interleaved.

$LEFT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow i.gets\ up \rightarrow LEFT_i)$

$RIGHT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.(i \oplus 1) \rightarrow i.puts\ down\ fork.(i \oplus 1) \rightarrow i.gets\ up \rightarrow RIGHT_i)$

$PHIL_i = LEFT_i \parallel RIGHT_i$

Concurrency: The Dining Philosophers: **Behaviour**

- A variation: each fork may be picked up and put down many times on each occasion that the philosopher sits down:

$$
\begin{aligned}
&LEFT_i = (i.sits\ down \rightarrow \\
&\qquad \mu X \bullet (i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow X \mid i.gets\ up \rightarrow LEFT_i))
\end{aligned}
$$

Concurrency: The Dining Philosophers: **Deadlock!**

- The constructed mathematical model reveals a serious danger.
  - all the philosophers get hungry at about the same time; they all sit down;
  - they all pick up their own forks;
  - they all reach out for the other fork — which isn’t there.
  - They will all inevitably starve.
    - each actor is capable of further action,
      - there is no action which any pair of them can agree to do next.
- Possible solutions:
  - One of the philosophers could always pick up the right fork first
    - if they could agree which one it should be.
  - The purchase of a single additional fork was ruled out for similar reasons.
  - The purchase of five more forks was much too expensive.

Concurrency: The Dining Philosophers: **Deadlock!**

- The solution:
  - a footman, whose duty it was to assist each philosopher into and out of his chair.
- His alphabet: $\bigcup_{i=0}^{4}\{i.sits\ down,\ i.gets\ up\}$
- This footman never allows more than four philosophers to be seated simultaneously.
- $FOOT_j$ defines the behaviour of the footman with j philosophers seated
  - $U = \bigcup_{i=0}^{4}\{i.gets\ up\}$ $D = \bigcup_{i=0}^{4}\{i.sits\ down\}$

$$FOOT_0 = (x : D \rightarrow FOOT_1)$$
$$FOOT_j = (x : D \rightarrow FOOT_{j+1} \mid y : U \rightarrow FOOT_{j-1}) \qquad \text{for } j \in \{1, 2, 3\}$$
$$FOOT_4 = (y : U \rightarrow FOOT_3)$$

- A college free of deadlock: $NEWCOLLEGE = (COLLEGE \parallel FOOT_0)$
- The dining philosophers problem – Edsger W. Dijkstra.
- The footman – Carel S. Scholten.

Concurrency: The Dining Philosophers: **Proof of absence of deadlock**

- We must prove that
- $(NEWCOLLEGE\ /\ s) \neq STOP$ for all $s \in traces(NEWCOLLEGE)$
- For an arbitrary trace s, in all cases
  - there is at least one event by which s can be extended and
    - still remain in $traces(NEWCOLLEGE)$.

$U = \bigcup_{i=0}^{4}\{i.gets\ up\}$ $D = \bigcup_{i=0}^{4}\{i.sits\ down\}$

$FOOT_0 = (x : D \rightarrow FOOT_1)$
$FOOT_j = (x : D \rightarrow FOOT_{j+1} \mid y : U \rightarrow FOOT_{j-1})$
$FOOT_4 = (y : U \rightarrow FOOT_3)$

- The number of seated philosophers
- $seated(s) = \#(s \upharpoonright D) - \#(s \upharpoonright U)$
- By L1 $s \upharpoonright (U \cup D) \in traces(FOOT_0)$, hence $seated(s) \leq 4$.
- If $seated(s) \leq 3$, at least one more philosopher can sit down
  - there is no deadlock.

L1. $traces(P \parallel Q) = \{ t \mid (t \upharpoonright \alpha P) \in traces(P) \wedge (t \upharpoonright \alpha Q) \in traces(Q) \wedge t \in (\alpha P \cup \alpha Q)^* \}$

Concurrency: The Dining Philosophers: **Proof of absence of deadlock**

- If $seated(s) = 4$,
  - consider the number of philosophers who are eating.
    - If this is nonzero, then an eating philosopher can always put down his left fork.
    - If it is zero, than no philosopher is eating,
      - consider the number of raised forks.
      - If this is three or less,
        - then one of the seated philosophers can pick up his left fork.
      - If there are four raised forks,
        - then the philosopher to the left of the vacant seat already has raised his left fork and can pick up his right one.
      - If there are five raised forks,
        - then at least one of the seated philosophers must be eating.

Concurrency: The Dining Philosophers: **Proof of absence of deadlock**

- This proof analyses a number of cases,
  - described in terms of the behaviour of this particular example.
- An alternative proof method
  - to explore all possible behaviours of the system to look for deadlock.
- In general, it is impossible.
- In the case of finite-state systems
  - it is sufficient to consider only those traces whose
    - length does not succeed a known upper bound on the number of states.
- The number of states of $(P \parallel Q)$ does not exceed
  - the product of the number of states of P and the number of states of Q.
- Since each philosopher has six states and each fork has four states
  - the total number of states of the *COLLEGE* does not exceed $6^5 \times 4^5 \approx 8$ million
- *NEWCOLLEGE* cannot have more states than the *COLLEGE* due to the footman alphabet.
- The number of traces is exceed $2^{8 \text{ million}}$.

Concurrency: The Dining Philosophers: **Infinite overtaking**

- A dining philosopher can be infinitely overtaken.
  - A seated philosopher has
    - a rather slow left arm, and
    - an extremely greedy left neighbour.
  - His left neighbour rushes in, sits down, rapidly picks up both forks, and spends a long time eating
    - before he can pick up his left fork.
  - Eventually the rapid neighbour puts down both forks, and leaves his seat.
  - But then he instantly gets hungry again, rushes in, sits down, and takes both forks,
    - before his right neighbour gets around to picking up the fork they share.
- Since this cycle may be repeated indefinitely,
  - a seated philosopher may never succeed in eating.

Concurrency: The Dining Philosophers: **Infinite overtaking**

- The effective solution is to buy more forks, and plenty of spaghetti.
- To guarantee that a seated philosopher will eventually eat
  - modify the behaviour of the footman:
    - having helped a philosopher to his seat he waits
      - until that philosopher has picked up both forks
      - before he allows either of his neighbours to sit down.

Concurrency: The Dining Philosophers: **Infinite overtaking**

- Infinite overtaking and fairness.
- Suppose the footman conceives an irrational dislike for one of his philosophers, and
  - persistently delays the action of escorting him to his chair,
  - even when the philosopher is ready to engage in that event.
- This is a possibility that cannot be described in our conceptual framework
  - we cannot distinguish it from the possibility that
    - the philosopher himself takes an indefinitely long time to get hungry.
- We have decided to ignore this problem, or
  - rather to delegate it to a different phase of design and implementation.
- It is an implementor's responsibility *to ensure that any desirable event that becomes possible will take place within an acceptable interval*.
- The implementor of a conventional high-level programming language has a similar obligation not to insert arbitrary delays into the execution of a program, even though the programmer has no way of enforcing or even describing this obligation.

Concurrency: **Change of symbol**

- A method of defining groups of processes with similar behaviour.
  - two collections of processes, philosophers and forks
- f Is a one-one function (injection) which maps the alphabet of P onto a set of symbols A
  - $f : aP \rightarrow A$
- The process $f(P)$ engages in the event $f(c)$ whenever P would have engaged in c:
  - $af(P) = f(aP)$
  - $traces(f(P)) = \{f^*(s) \mid s \in traces(P)\}$

f^* maps a sequence of symbols in A^* to a sequence in B^* by applying f to each element of the sequence

Concurrency: **Change of symbol**

X1. After a few years, the price of everything goes up.

- To represent the effect of inflation, we define a function f :

$$f(in2p) = in10p \qquad f(large) = large$$
$$f(in1p) = in5p \qquad f(small) = small$$
$$f(out1p) = out5p$$

- The new vending machine is
  - $NEWVMC = f(VMC)$

$$VMC = (in2p \rightarrow (large \rightarrow VMC \mid small \rightarrow out1p \rightarrow VMC)$$
$$\mid in1p \rightarrow (small \rightarrow VMC \mid in1p \rightarrow (large \rightarrow VMC \mid in1p \rightarrow STOP)))$$

X2. A counter behaves like CT_0, except that it moves *right* and *left* instead of *up* and *down*:

  - $LR_0 = f(CT_0)$

$$f(up) = right \qquad f(down) = left \qquad f(around) = around$$

$$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow CT_0)$$
$$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n)$$

Concurrency: **Change of symbol**

X3. A counter moves *left, right*, *up* or *down* on an infinite board with boundaries at the left and at the bottom

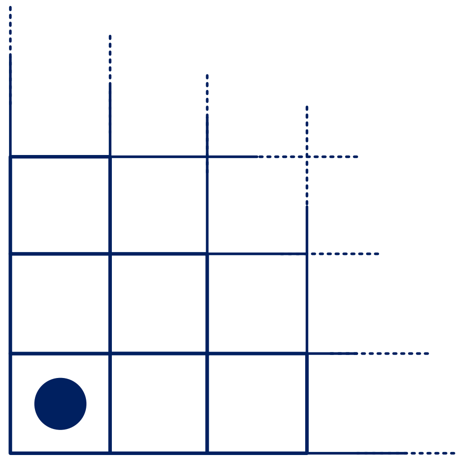
$LR_0 = (right \rightarrow LR_1 \mid around \rightarrow LR_0)$
$LR_{n+1} = (right \rightarrow LR_{n+2} \mid left \rightarrow LR_n)$

$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow CT_0)$
$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n)$

- It starts at the bottom left corner.
- On this square alone, it can turn *around*.
- Vertical and horizontal movements can be modelled as independent actions of separate processes;

  but *around* requires simultaneous participation of both

- $LRUD = LR_0 \parallel CT_0$



Concurrency: **Change of symbol**

X4. Connect two instances of *COPYBIT* in series, so that

each bit output by the first is simultaneously input by the second.

- The new events used for internal communication: *mid.0* and *mid.1*
- The functions *f* and *g* to change the output of one process and the input of the other:

$$
\begin{aligned}
&f(out.0) = g(in.0) = mid.0 \\
&f(out.1) = g(in.1) = mid.1 \\
&f(in.0) = in.0, \qquad f(in.1) = in.1 \\
&g(out.0) = out.0, \qquad g(out.1) = out.1
\end{aligned}
$$

- The synchronised communication of binary digits on a channel
  - $CHAIN2 = f(COPYBIT) \parallel g(COPYBIT)$
- Each output of *0* or *1* by the left operand of ‖ is the very same event (*mid.0* or *mid.1*) as the input of the same *0* or *1* by the right operand.

$$COPYBIT = \mu X \bullet (in.0 \rightarrow out.0 \rightarrow X \mid in.1 \rightarrow out.1 \rightarrow X)$$

$$CHAIN2 = f(COPYBIT) \parallel g(COPYBIT)$$

Concurrency: **Change of symbol**

- The left operand offers no choice of which value is transmitted on the channel.
- The right operand is prepared to engage in either of the events *mid.0* or *mid.1*.
  - The outputting process that determines which of these two events occur.
- The internal communications *mid.0* and *mid.1*
  - are in the alphabet of the composite processes, and
  - can be observed (or even perhaps controlled) by its environment.
  - Ignoring or concealing such internal events may introduce nondeterminism.

$f(out.0) = g(in.0) = mid.0$
$f(out.1) = g(in.1) = mid.1$
$f(in.0) = in.0, \quad f(in.1) = in.1$
$g(out.0) = out.0, \quad g(out.1) = out.1$

$DD = (setorange \rightarrow O \mid setlemon \rightarrow L)$
$O = (orange \rightarrow O \mid setlemon \rightarrow L \mid setorange \rightarrow O)$
$L = (lemon \rightarrow L \mid setorange \rightarrow O \mid setlemon \rightarrow L)$

Concurrency: **Change of symbol**

X5. The behaviour of a Boolean variable used by a computer program.

- The events in its alphabet are
  - *assign0* — assignment of value zero to the variable
  - *assign1* — assignment of value one to the variable
  - *fetch0* — access of the value of the variable at a time when it is zero
  - *fetch1* — access of the value of the variable at a time when it is one
- The behaviour of the variable is similar to that of the drinks dispenser
  - $BOOL = f(DD)$
- where the definition of *f* is a trivial exercise.
- The Boolean variable refuses to give its value until after a value has been first assigned.

Concurrency: **Change of symbol**

- The tree picture of *f(P)* may be constructed from the tree picture of *P* by
  - applying the function *f* to the labels on all the branches.
    - This transformation preserves the structure of the tree
      - Because *f* is a one–one function
- A picture of *NEWVMC* :

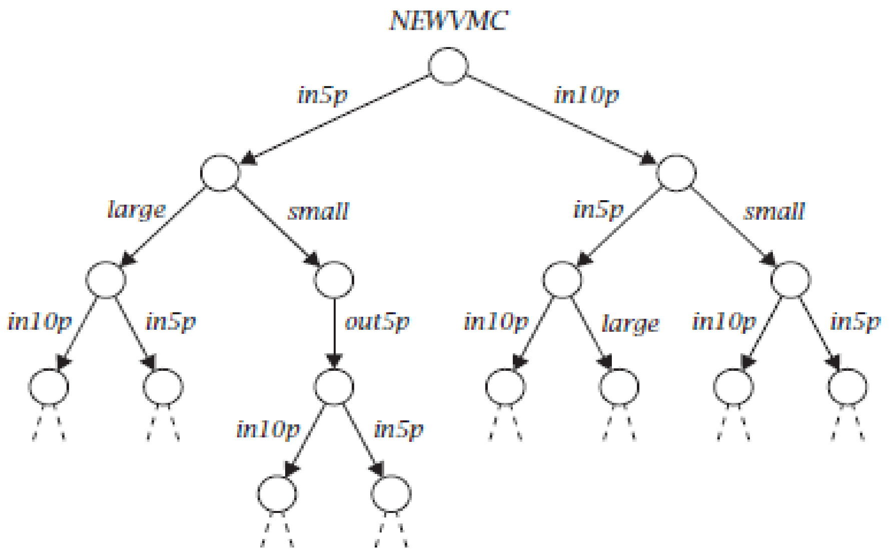


Figure 2.8

Concurrency: Change of symbol: **Laws**

- Change of symbol by application of a one-one function does not change
  - the structure of the behaviour of a process.
  - function application distributes through all the other operators
- Notation:

$f(B) = \{f(x) \mid x \in B\}$
f^{-1} is the inverse of f
$f \circ g$ is the composition of f and g
f^* is the *sequence* function

f^* maps a sequence of symbols in A^* to a sequence in B^* by applying f to each element of the sequence

Concurrency: Change of symbol: **Laws**

- After change of symbol, *STOP* still performs no event from its changed alphabet:

L1. $f(STOP_A) = STOP_{f(A)}$

- The symbols offered for selection are changed, and the subsequent behaviour is changed

L2. $f(x : B \rightarrow P(x)) = (y : f(B) \rightarrow f(P(f^{-1}(y))))$

  - P is a process depending on selection of some x from the set B.
  - But the variable y on the right-hand side is selected from the set $f(B)$.
  - The corresponding event for P is $f^{-1}(y)$, which is in B (since $y \in f(B)$).
  - The behaviour of P after this event is $P(f^{-1}(y))$, and
    - the actions of this process must continue to be changed by application of f.

Concurrency: Change of symbol: Laws

- Change of symbol distributes through parallel composition:

L3. $f(P \parallel Q) = f(P) \parallel f(Q)$

- Change of symbol distributes in a more complex way over recursion:

L4. $f(\mu X : A \bullet F(X)) = (\mu Y : f(A) \bullet f(F(f^{-1}(Y))))$

- The validity of the recursion on the left-hand side requires that F is a function which
  - takes as argument a process with alphabet A, and
  - delivers a process with the same alphabet.
- On the right-hand side,
  - Y is a variable ranging over processes with alphabet $f(A)$, and
  - cannot be used as an argument to F until its alphabet has been changed back to A.
- After applying the inverse function f^{-1} to Y
  - $F(f^{-1}(Y))$ has alphabet A, so
  - an application of f will transform the alphabet to $f(A)$,
    - thus ensuring the validity of the recursion on the right-hand side of the law.

Concurrency: Change of symbol: **Laws**

- The composition of two changes of symbol is
  - the composition of the two symbol-changing functions

L5. $f(g(P)) = (f \circ g)\ (P)$

- The traces of a process after change of symbol:

L6. $traces(f(P)) = \{\ f^*(s)\ \mid\ s \in traces(P)\ \}$

- The element-wise changing "after":

L7. $f(P)\ /\ f^*(s) = f(P\ /\ s)$

Concurrency: Change of symbol: **Process labelling**

- Change of symbol is for constructing *groups of similar processes* which
  - operate concurrently,
  - provide identical services to their common environment,
  - do not interact with each other.
- They must all have different and mutually disjoint alphabets.
  - each process is labelled by a different name
    - process P labelled by l -- $l : P$
    - each event of a labelled process is labelled by its name.
      - event x is labelled by $l.x$
- The function to define $l : P$ is
  - $fl(x) = l.x$ for all x in αP
- The labelled process $l : P = f_l(P)$

Concurrency: Change of symbol: **Process labelling**

X1. A pair of vending machines standing side by side

- $(left : VMS) \parallel (right : VMS)$

- Their alphabets are disjoint
  - every event is labelled by the name of its machine.

- If these machines is not named
  - every event would require participation of both of them,
    - they would be indistinguishable from a single machine

- $(VMS \parallel VMS) = VMS$

$VMS = (coin \rightarrow (choc \rightarrow VMS))$

$\alpha BOOL$ = {assign0, assign1, fetch0, fetch1}

Concurrency: Change of symbol: **Process labelling**

X2. The behaviour of a Boolean variable is process *BOOL*.

- The behaviour of a block of program is process *USER*:
  - assigns and accesses the values of two Boolean variables named *b* and *c*.
- *αUSER* includes
  - *b.assign.0* — to assign value zero to *b*
  - *c.feth.1* — to access the current value of *c* when it is one
- The *USER* process runs in parallel with its two Boolean variables
  - $b : BOOL \parallel c : BOOL \parallel USER$
- Inside the *USER* program, the following effects may be achieved

  $b := false \; ; P$ by $(b.assign.0 \rightarrow P)$

  $b := \neg c \; ; P$ by $(c.fetch.0 \rightarrow b.assign.1 \rightarrow P \mid c.fetch.1 \rightarrow b.assign.0 \rightarrow P)$
- The choice of the current value of the variable (*fetch.0* and *fetch.1*) affects the subsequent behaviour of the *USER*.
- Further notation for the single assignment: $b := false$ instead of $b := false \; ; P$

$COPYBIT = \mu X \bullet (in.0 \rightarrow out.0 \rightarrow X \mid in.1 \rightarrow out.1 \rightarrow X)$

$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow CT_0)$
$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n)$

Concurrency: Change of symbol: **Process labelling**

X3. A *USER* process

- has two count variables named l and m initialised to 0 and 3 respectively,
- increments each variable by $l.up$ or $m.up$,
- decrements it (when positive) by $l.down$ and $m.down$.
- tests for zero by the events $l.around$ and $m.around$.
- The process CT can be used after appropriate labelling by l and by m
- Within the *USER* process the following effects can be achieved

$(l : CT_0 \parallel m : CT_3 \parallel USER)$

$(m := m + 1 \,;\, P)$ by $(m.up \rightarrow P)$

if $l = 0$ **then** P **else** Q by $(l.around \rightarrow P \mid l.down \rightarrow l.up \rightarrow Q)$

- The test for zero:
  - attempting $l.down$ at the same time as attempting $l.around$.
    - if the value is zero, $l.around$ is selected; if non–zero, the other.
      - In the latter case, the value of the count must be restored.

Concurrency: Change of symbol: **Process labelling**

$(m := m + l \,;\, P)$

- Addition is implemented by *ADD*:

$$
\begin{aligned}
&ADD = DOWN_0\\
&DOWN_i = (l.down \rightarrow DOWN_{i+1} \mid l.around \rightarrow UP_i)\\
&UP_0 = P\\
&UP_{i+1} = l.up \rightarrow m.up \rightarrow UP_i
\end{aligned}
$$

- The $DOWN_i$ processes discover the initial value of l by decrementing it to zero.
- The UP_i processes then add the discovered value to both m and to l, thereby restoring l to its initial value and adding this value to m.

Concurrency: Change of symbol: **Multiple labelling**

- Multiple labelling: each event may take any label l from a set L.
- $(L : P)$ is a process which behaves exactly like process P,
  - except that it engages in event $l.c$ (where $l \in L$ and $c \in \alpha P$) if P would have done c.
  - The choice of label l is made independently by the environment of $(L : P)$.

X1. A lackey is a junior footman, who helps his single master to and from his seat, and stands behind his chair while he eats

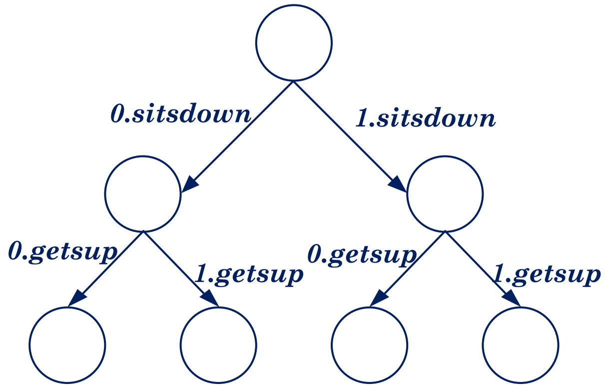
- $\alpha LACKEY = \{sits\ down, gets\ up\}$ $LACKEY = (sits\ down \rightarrow gets\ up \rightarrow LACKEY)$
- The lackey may share his services among five masters (but serving only one at a time):

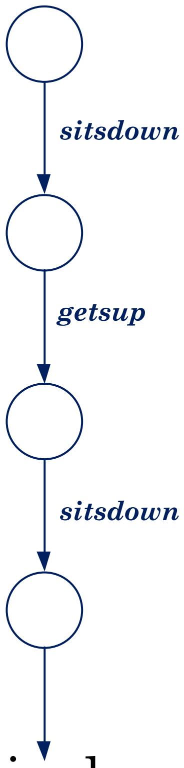
$$L = \{0, 1, 2, 3, 4\} \qquad SHARED\ LACKEY = (L : LACKEY)$$

- The shared lackey could be employed to protect the dining philosophers from deadlock when the footman is on holiday.
  - The philosophers may go hungrier during the holiday, since only one of them is allowed to the table at a time.

Concurrency: Change of symbol: **Multiple labelling**

- If L contains more than one label, the tree picture of $L : P$ is similar to that for P,
  - it is much more bushy because there are more branches leading from each node.
- The picture of the *LACKEY* is a single trunk with no branches.
- The picture of *{0, 1}* : *LACKEY* is a complete binary tree.





- The tree for the *SHARED LACKEY* is even more bushy
- In general, multiple labelling can be used to share the services of a single process among a number of other labelled processes, provided that the set of labels is known in advance.