

Chapter 6. Applications of CCS

This chapter contains several examples of concurrent and communicating systems. For each of these systems we give CCS expressions describing their specifications and possible implementations. We also give proofs of correctness of implementations with respect to specifications both by means of $=$ and \approx_o .

1 Summary of the module

- Chapter 1 uses examples to motivate the theory, and to informally introduce CCS.
- Chapter 2 formally defines the calculus, i.e. its syntax and semantics. A system specification is then written as an agent expression in the calculus, and its behaviour is analysed by applying SOS rules.
- Chapters 3–5, develop the algebraic and operational theories, which include three equivalence relations and their properties. The theories allow us to manipulate agent expressions, changing one into another equivalent one, and to reason about whether one agent (implementation) is equivalent to another one (specification) by comparing their transition graphs.

This chapter applies the theory to a number of examples which are not quite trivial. The examples will show how to apply CCS in system development from a specification to possible implementations.

2 A Scheduler

Suppose that a set of agents P_i , $1 \leq i \leq n$, is to be scheduled in performing a certain task, i.e. each agent P_i wishes to perform the task repeatedly, and a scheduler is required to ensure that

- they begin the task in a cyclic order starting with P_1 ,
- the different task-performances need not exclude each other in time: e.g. P_2 can begin before P_1 finishes.
- but each agent finishes one performance before it begins another.

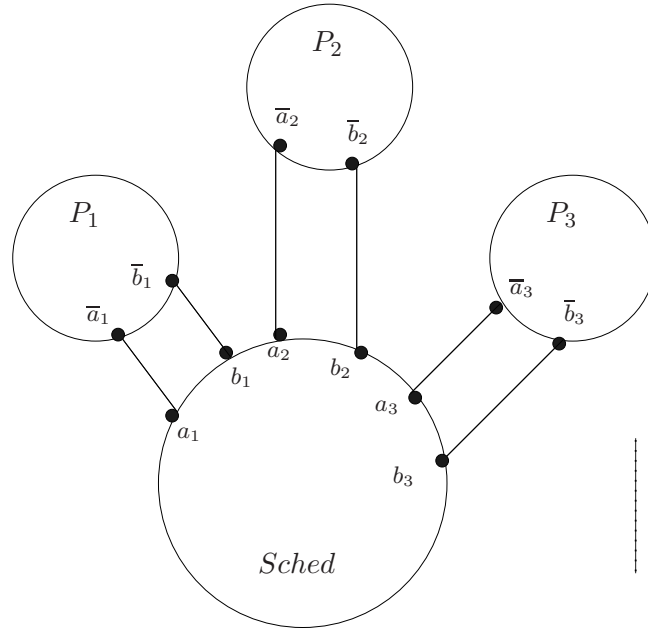
Assume that P_i requests task initiation by $\overline{a_i}$, and signals completion by $\overline{b_i}$. Thus, the scheduler has sort $\tilde{a} \cup \tilde{b}$, where

$$\tilde{a} = \{a_1, \dots, a_n\} \quad \tilde{b} = \{b_1, \dots, b_n\}$$

Requirements

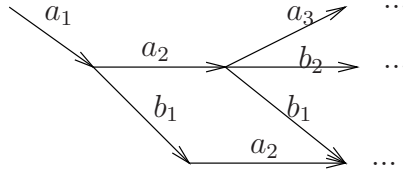
1. The scheduler must perform a_1, \dots, a_n cyclically, starting with a_1
2. The scheduler must perform a_i and b_i alternately, for each i .

Flow graph of the scheduler

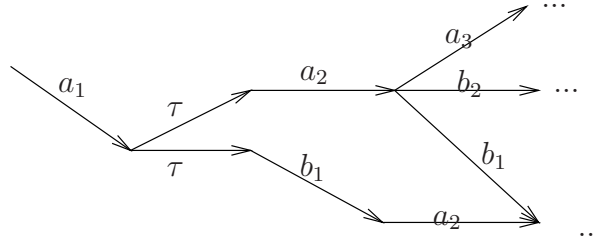


Vagueness of the informal specification

- A scheduler which performs a fixed sequence - say $a_1b_1a_2b_2\dots$, may satisfy the requirement conditions 1 and 2. But it is not good enough since there are many other sequences compatible with conditions 1 and 2.
- We may expect (or we may interpret the two conditions as such that) *Sched* to have the following (partial) derivation tree.



- We may also consider (interpret the two conditions as such that *Sched* has) the following tree.



Which of these interpretation(s) are acceptable? The second proposal has the following drawbacks.

- The upper τ action represents an autonomous commitment by the scheduler to do a_2 before b_1 , and the lower τ action represents the opposite commitment.

- Such an nondeterministic scheduler runs the risk of causing deadlock. For example, if after it starts, agent P_1 needs the final output of P_2 to complete, the system as a whole would deadlock if the lower τ is fired by the scheduler.

Formal specification

A formal specification will help to rule out vagueness and to avoid errors at early stage in the system development. The specification *Schedspec* is written in terms of a set of states:

For i ($1 \leq i \leq n$) and $X \subseteq \{1, \dots, n\}$, let *Schedspec*(i, X) be the agent such that

1. It is P_i 's turn to initiate next;
2. The agents $\{P_j : j \in X\}$ are currently performing the task.

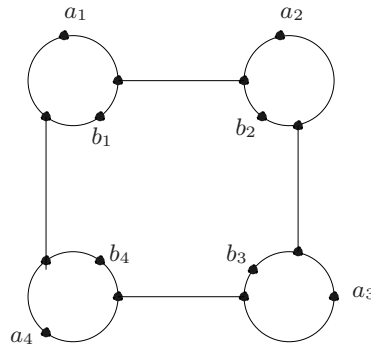
In the state *Schedspec*(i, X), any P_j ($j \in X$) may terminate; also P_i may initiate provided $i \notin X$.

$$\begin{aligned} \text{Schedspec} &\stackrel{\text{def}}{=} \text{Schedspec}(1, \emptyset) \\ (i \in X) \quad \text{Schedspec}(i, X) &\stackrel{\text{def}}{=} \sum_{j \in X} b_j. \text{Schedspec}(i, X - \{j\}) \\ (i \notin X) \quad \text{Schedspec}(i, X) &\stackrel{\text{def}}{=} a_i. \text{Schedspec}(i + 1, X \cup \{i\}) \\ &\quad + \sum_{j \in X} b_j. \text{Schedspec}(i, X - \{j\}) \end{aligned}$$

where we adopt the convention that $i + 1$ is calculated modulo n , namely if $i = n$ then $i + 1$ is actually 1.

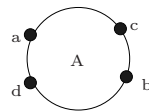
Implementing the scheduler

Build the scheduler, *Sched*, as a ring of n cells each linked to one of the agents P_i :



Let us first attempt at choosing a cycler cell

$$A \stackrel{\text{def}}{=} a.c.b.d.A$$



- It appears to work: e.g. after a_1 and a_2 , P_3 may initiate, and either P_1 or P_2 may complete.
- It does not quite work: e.g. for $n = 4$, after P_1, \dots, P_4 have all initiated but none has terminated. Then P_4 cannot terminate before P_1 has terminated – and this violated the specification since $Schedspec(1, \{1, 2, 3, 4\})$ allows any b_i , even b_4 , as the next action.
- Such a small bug may cause deadlock if P_4 and P_1 had agreed that P_4 should terminate first.

Exercise 1 What slight modification to $Schedspec$ would yield a specification satisfied by this scheduler? You cannot expect to be certain of the answer, but make an intelligent guess.

Second attempt at the design

- Modify the cell so that b and d can occur in either order:

$$A \stackrel{def}{=} a.c.(b.d.A + d.b.A)$$

or, for convenience of reference

$$\begin{aligned} A &\stackrel{def}{=} a.C & C &\stackrel{def}{=} c.E \\ E &\stackrel{def}{=} b.D + d.B \\ B &\stackrel{def}{=} b.A & D &\stackrel{def}{=} d.A \end{aligned}$$

- Define the relabelling functions, for $i = 1, \dots, n$

$$f_i = (a_i/a, b_i/b, c_i/c, \overline{c_{i-1}}/d)$$

where c_0 is defined to be c_n .

- Let

$$\begin{aligned} A_i &\stackrel{def}{=} A[f_i] & C_i &\stackrel{def}{=} C[f_i] \\ E_i &\stackrel{def}{=} E[f_i] & B_i &\stackrel{def}{=} B[f_i] \\ D_i &\stackrel{def}{=} D[f_i] \end{aligned}$$

- The scheduler is given as

$$Sched \stackrel{def}{=} (A_1 | D_2 | \dots | D_n) \setminus \tilde{c}$$

where $\tilde{c} = \{c_1, \dots, c_n\}$. Or write it as

$$Sched = (A_1 | \Pi_{i \neq 1} D_i) \setminus \tilde{c}$$

- The meaning of the states of the i^{th} cell:

A_i	means	P_i 's turn, P_i ready to initiate
C_i	means	not P_i 's turn, P_i not ready to initiate
E_i	means	not P_i 's turn, P_i not ready to initiate
B_i	means	P_i 's turn, P_i not ready to initiate
D_i	means	not P_i 's turn, P_i ready to initiate

Verification

We need to prove

$$Schedspec = Sched$$

Since $Schedspec$ and $Sched$ are both stable, it is enough to show (see Proposition 5.10)

$$Schedspec \approx Sched$$

Proof outline

- Based on the meaning of $Schedspec(i, X)$, and that of A_i , B_i , etc., we define the following states of $Sched$, for each i and $X \subseteq \{1, \dots, n\}$

$$Sched(i, X) \stackrel{def}{=} \begin{cases} (B_i | \prod_{j \notin X} D_j | \prod_{j \in X - \{i\}} E_j) \backslash \tilde{c} & \text{if } i \in X \\ (A_i | \prod_{j \notin X \cup \{i\}} D_j | \prod_{j \in X} E_j) \backslash \tilde{c} & \text{if } i \notin X \end{cases}$$

- Then prove that the following is a WB up to \approx

$$\{(Schedspec(i, X), Sched(i, X)) : 1 \leq i \leq n, X \subseteq \{1, \dots, n\}\}$$

Note that $Schedspec = Schedspec(1, \emptyset)$, $Sched = Sched(1, \emptyset)$.

Thus $Schedspec \approx Sched$.

And $Schedspec = Sched$ as both agents are stable.

- The proof of WB up to \approx for \mathcal{S} is mainly a matter of case analysis by using the definition.
- However, the following lemma is helpful.

Lemma For $i = 1, \dots, n$

1. $(C_i | D_{i+1}) \backslash c_i \approx (E_i | A_{i+1}) \backslash c_i$
2. $(C_i | E_{i+1}) \backslash c_i \approx (E_i | B_{i+1}) \backslash c_i$

Read Sections 5.4 and 5.5 of Milner's book for further detail.

Exercise 2 Prove the lemma.

3 A Filter System

This system was the subject of 2000 exam question.

1. Consider agent $Filter_K$, a one-place buffer-like agent that repeatedly accepts integers on channel in and outputs them on channel \overline{out} provided that they belong to the set K . If input integers are not members of K , they are not output. In the value passing CCS, write a specification of $Filter_K$.
2. A set of integers K of cardinality n is implemented by an agent $Tape(S)$, where S is any fixed sequence of all elements of K . Only one element of S can be accessed by the user at any time. The first element of S is accessible in the initial state of the tape agent. In order to access the next element of S the user needs to perform action \overline{up} . In order to access the subsequent elements of S the user needs to perform further actions \overline{up} . When the tape agent is not in the initial state, namely when the l th element of S is accessible for $1 < l \leq n$, the user can access the previous element of S , and similarly the preceding elements of S , by performing a suitable number of actions \overline{down} . When an element x of S is accessible, the user can read x by action $read(x)$. In the value passing CCS, write a specification of $Tape(S)$.
3. Construct an agent $FILTER_K$ by combining $Tape(S)$, where S is any fixed sequence of all elements of K , and agents $Checker$ and $ReceiveSend$, which you need to define, such that $FILTER_K$ implements $Filter_K$. Informally, $ReceiveSend$ receives an integer x on in , send it to $Checker$ to verify the membership of K , and outputs x on \overline{out} if instructed by $Checker$. Agent $Checker$ receives an integer x from $ReceiveSend$, uses $Tape(S)$ to check if x is an element of S (S is a fixed sequence of elements of K), moves $Tape(S)$ to its original state, and finally notifies $ReceiveSend$ of the result of the check. In your construction, you will need to use relabelling, restriction and parallel composition combinators as well as other combinators.
4. Draw a flow diagram for your construction in (3).
5. Argue formally, for example using CCS equational reasoning, the correctness of your construction in (3).

A possible solution is the following.

1.

$$\begin{aligned} Filter_K &\stackrel{def}{=} in(x).Filter_K(x) \\ Filter_K(x) &\stackrel{def}{=} \text{if } x \in K \text{ then } \overline{out}(x).Filter_K \text{ else } Filter_K \end{aligned}$$

2. The i th element of S is denoted by $S(i)$ for $1 \leq i \leq n \geq 2$.

$$\begin{aligned} Tape(S) &\stackrel{def}{=} Tape(S, 1) \\ Tape(S, 1) &\stackrel{def}{=} \overline{read}(S(1)).Tape(S, 1) + up.Tape(S, 2) \\ Tape(S, i) &\stackrel{def}{=} \overline{read}(S(i)).Tape(S, i) + up.Tape(S, i + 1) + down.Tape(S, i - 1) \\ &\quad \text{for } 1 < i < n \\ Tape(S, n) &\stackrel{def}{=} \overline{read}(S(n)).Tape(S, n) + down.Tape(S, n - 1) \end{aligned}$$

3. Firstly, we define *ReceiveSend*.

$$\begin{aligned}
\textit{ReceiveSend} &\stackrel{\text{def}}{=} \textit{in}(x).\textit{RS}(x) \\
\textit{RS}(x) &\stackrel{\text{def}}{=} \bar{g}(x).\textit{RS}'(x) \\
\textit{RS}'(x) &\stackrel{\text{def}}{=} h.\overline{\textit{out}}(x).\textit{RS}'' + j.\textit{RS}'' \\
\textit{RS}'' &\stackrel{\text{def}}{=} l.\textit{ReceiveSend}
\end{aligned}$$

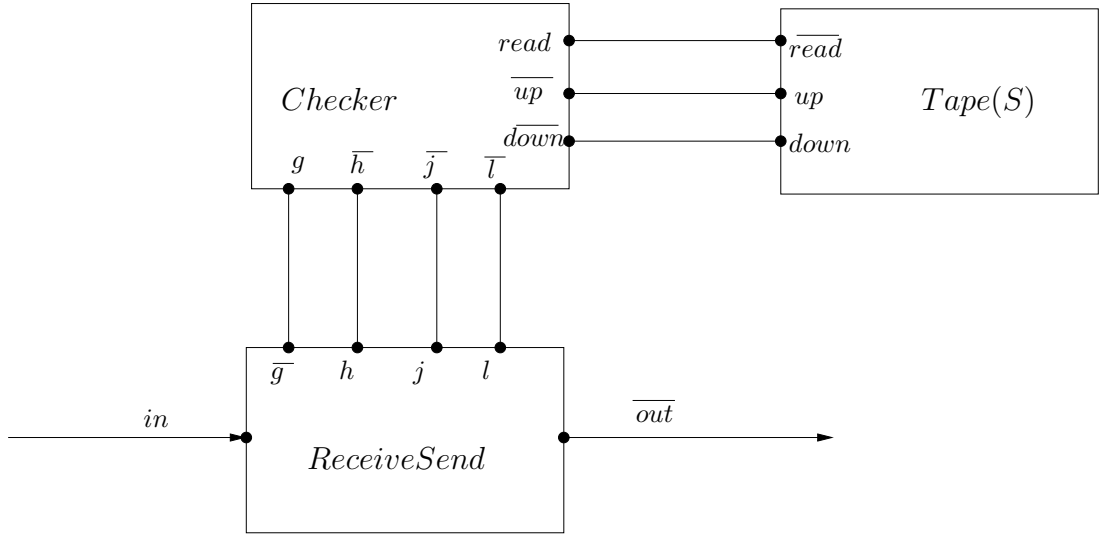
Next, we define *Checker*.

$$\begin{aligned}
\textit{Checker} &\stackrel{\text{def}}{=} g(x).\textit{Checker}(x) \\
\textit{Checker}(x) &\stackrel{\text{def}}{=} \textit{read}(y).\textit{Checker}_1(x, y) \\
\textit{Checker}_i(x, y) &\stackrel{\text{def}}{=} \text{if } x = y \text{ then } \bar{h}.\textit{End}_i \text{ else } \overline{\textit{up}}.\textit{read}(z).\textit{Checker}_{i+1}(x, z) \\
&\quad \text{for } 1 \leq i < n \\
\textit{Checker}_n(x, y) &\stackrel{\text{def}}{=} \text{if } x = y \text{ then } \bar{h}.\textit{End}_n \text{ else } \bar{j}.\textit{End}_n \\
\textit{End}_i &\stackrel{\text{def}}{=} \text{if } i > 1 \text{ then } \overline{\textit{down}}.\textit{End}_{i-1} \text{ else } \bar{l}.\textit{Checker}
\end{aligned}$$

Assuming that S is a particular sequence of elements of K , the system \textit{FILTER}_K is defined as follows.

$$\textit{FILTER}_K \stackrel{\text{def}}{=} (\textit{ReceiveSend}|\textit{Checker}|\textit{Tape}(S)) \setminus \{g, h, j, \textit{read}, \textit{up}, \textit{down}\}$$

4.



5. Let $L = \{g, h, j, \textit{read}, \textit{up}, \textit{down}\}$. We use CCS equational reasoning to prove correctness. For this we need to be dealing with pure CCS expressions. Thus, as a first step of the proof we should convert value passing CCS expressions to pure CCS expressions. In order to avoid rewriting all above agent definitions, we stop short of using proper pure CCS notation but use value passing-like notation. So, for example, instead of \textit{in}_a we use $\textit{in}(a)$ and instead of $\textit{Checker}_a$ we use $\textit{Checker}(a)$.

By the Expansion Law and the 1st τ -law we obtain

$$(\textit{ReceiveSend}|\textit{Checker}|\textit{Tape}(S)) \setminus L = \textit{in}(x).(\textit{RS}'(x)|\textit{Checker}(x)|\textit{Tape}(S)) \setminus L$$

The RHS is actually equal to

$$\sum_{a \in \mathbb{Z}} in(a).(RS'(a)|Checker(a)|Tape(S)) \setminus L$$

For each a $(RS'(a)|Checker(a)|Tape(S)) \setminus L$ performs a sequence of τ actions corresponding to the synchronisations of checker reading elements y of S and checking if $a = y$. There are two cases. If a is indeed an element of S and $a = S(k)$, for $1 \leq k \leq n$, then the system would be in the state $(RS'(a)|\bar{h}.End_k|Tape(S, k)) \setminus L$. Now, the communication on h takes place, then the systems performs $\overline{out}(a)$ and, finally, $k - 1$ actions τ to move the tape agent back to its initial state. After this there is one more τ (communication on l) by which *Checker* tells *ReceiveSend* that it and the tape agent are back in their initial states. Now, *ReceiveSend* can receive the next value on *in*. So, if $a \in K$ and it is the k th element of S , then we have

$$(RS'(a)|Checker(a)|Tape(S)) \setminus L \xrightarrow{(\tau)^{2k-1}} \xrightarrow{\tau \overline{out}(a)} \xrightarrow{(\tau)^{k-1}} \xrightarrow{\tau} (ReceiveSend|Checker|Tape(S)) \setminus L.$$

If a is not a member of S , then

$$(RS'(a)|Checker(a)|Tape(S)) \setminus L \xrightarrow{(\tau)^{2n-1}} \xrightarrow{(\tau)^{n-1}} \xrightarrow{\tau} (ReceiveSend|Checker|Tape(S)) \setminus L.$$

Thus, $\sum_{a \in \mathbb{Z}} in(a).(RS'(a)|Checker(a)|Tape(S)) \setminus L$ equals

$$\begin{aligned} & \sum_{a \in \mathbb{Z}, a=S(k)} in(a).\tau^{2k}.\overline{out}(a).\tau^k.(ReceiveSend|Checker|Tape(S)) \setminus L + \\ & \sum_{a \in \mathbb{Z}, a \notin S} in(a).\tau^{3n}.(ReceiveSend|Checker|Tape(S)) \setminus L \\ = & \sum_{a \in \mathbb{Z}, a=S(k)} in(a).\overline{out}(a).(ReceiveSend|Checker|Tape(S)) \setminus L + \\ & \sum_{a \in \mathbb{Z}, a \notin S} in(a).(ReceiveSend|Checker|Tape(S)) \setminus L \\ = & \sum_{a \in \mathbb{Z}} in(a).(\text{ if } a \in K \text{ then } \overline{out}(a).(ReceiveSend|Checker|Tape(S)) \setminus L \\ & \text{ else } (ReceiveSend|Checker|Tape(S)) \setminus L) \end{aligned}$$

Consider the equation

$$X = \sum_{a \in \mathbb{Z}} in(a).(\text{ if } a \in K \text{ then } \overline{out}(a).X \text{ else } X)$$

We notice that $Filter_K$ and $FILTER_K$ are solutions of this equation. Since X above is both sequential and guarded, the equation has a unique solution, thus $Filter_K = FILTER_K$.

Alternatively, one may find an observational congruence relation R such that $(Filter_K, FILTER_K) \in R$.

4 Proving the Jobshop Correct

We now turn to a proof of the equation

$$Jobshop = Strongjobber|Strongjobber$$

Firstly, we rewrite the definitions for specification and implementation agents. Then, we proceed to verification.

Specification

$$\begin{aligned}
Jobshopspec &\stackrel{def}{=} Strongjobber|Strongjobber \\
Strongjobber &\stackrel{def}{=} in(j).Doing(j) \\
Doing(j) &\stackrel{def}{=} \overline{out}(done(j)).Strongjobber
\end{aligned}$$

Implementation

$$\begin{aligned}
Jobshop &\stackrel{def}{=} (Jobber|Jobber|Ham|Mal)\backslash L \\
L &= \{geth, puth, getm, putm\} \\
Ham &\stackrel{def}{=} geth.Ham' \\
Ham' &\stackrel{def}{=} puth.Ham \\
Mal &\stackrel{def}{=} getm.Mal' \\
Mal' &\stackrel{def}{=} putm.Mal \\
Jobber &\stackrel{def}{=} in(j).Start(j) \\
Start(j) &\stackrel{def}{=} \text{if } easy(j) \text{ then } Finish(j) \\
&\quad \text{else if } hard(j) \text{ then } Useh(j) \\
&\quad \text{else } Usetool(j) \\
Usetool(j) &\stackrel{def}{=} Useh(j) + Usem(j) \\
Useh(j) &\stackrel{def}{=} \overline{geth}.Usingh(j) \\
Usingh(j) &\stackrel{def}{=} \overline{puth}.Finish(j) \\
Usem(j) &\stackrel{def}{=} \overline{getm}.Usingm(j) \\
Usingm(j) &\stackrel{def}{=} \overline{putm}.Finish(j) \\
Finish(j) &\stackrel{def}{=} \overline{out}(done(j)).Jobber
\end{aligned}$$

Verification

As both *Jobshop* and *Jobshopspec* are stable, it is enough to show that $Jobshop \approx Jobshopspec$. Thus, only one of them is needed in a WB up to \approx . We exhibit the relation \mathcal{S} below, which can be proven to be a WB up to \approx . Note, that *S* and *D* stand for *Strongjobber* and *Doing* respectively.

The pairs in \mathcal{S}

- (1) $(J|J|H|M)\backslash L, \quad S|S$
- (2) $(J|St(j)|H|M)\backslash L, \quad S|D(j)$
- (3) $(J|Uh(j)|H'|M)\backslash L, \quad S|D(j)$
- (4) $(J|Um(j)|H|M')\backslash L, \quad S|D(j)$
- (5) $(J|F(j)|H|M)\backslash L, \quad S|D(j)$
- (6) $(St(j')|St(j)|H|M)\backslash L, \quad D(j')|D(j)$
- (7) $(St(j')|Uh(j)|H'|M)\backslash L, \quad D(j')|D(j)$
- (8) $(St(j')|Um(j)|H|M')\backslash L, \quad D(j')|D(j)$
- (9) $(St(j')|F(j)|H|M)\backslash L, \quad D(j')|D(j)$
- (10) $(Uh(j')|Um(j)|H'|M')\backslash L, \quad D(j')|D(j)$
- (11) $(Uh(j')|F(j)|H'|M)\backslash L, \quad D(j')|D(j)$
- (12) $(Um(j')|F(j)|H|M')\backslash L, \quad D(j')|D(j)$
- (13) $(F(j')|F(j)|H|M)\backslash L, \quad D(j')|D(j)$

Notes on the proof

1. Just check the two conditions of the definition WB up to \approx for each of the 13 pairs.
2. Note that the specifications are written in the full value-passing calculus. Thus, we should think of $in(j).Start(j)$ as a short for

$$\sum_j in_j.Start(j)$$

where j ranges over all jobs. So *Jobber* has the transition

$$Jobber \xrightarrow{in_j} Start(j)$$

for any job j .

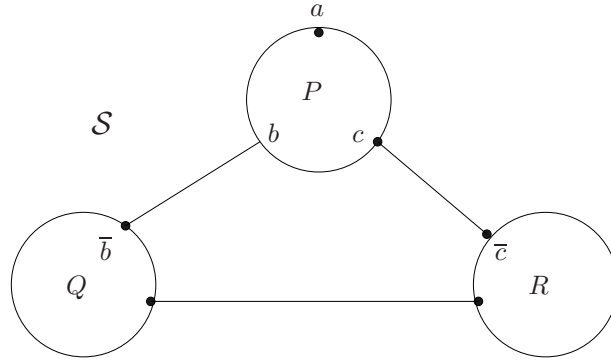
3. See Section 5.6 of Milner's book for more details.

5 Systems with Evolving Structure

Consider a system

$$S \equiv (P|Q|R)\backslash L$$

with $P : \{a, b, c\}$, and $b, c \in L$ but $a \notin L$.



Consider an action $P \xrightarrow{a} P'$. Then $S \xrightarrow{a} S'$, where

$$S' \equiv (P'|Q|R) \setminus L$$

Dynamic variations of structure

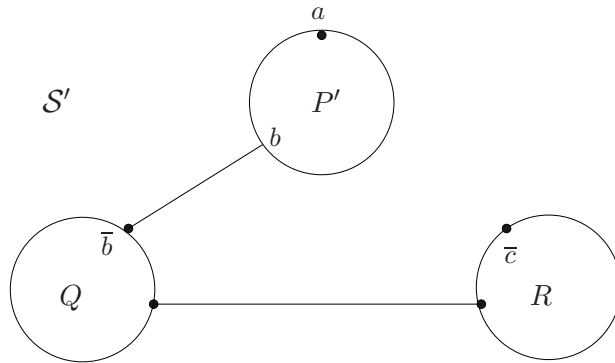
1. **No change:** if the sort of P' is the same as that of P ; e.g.

$$P \stackrel{\text{def}}{=} a.P \text{ so } P' = P$$

2. **Loss of an arc:** if $P' : \{a, b\}$; e.g.

$$P \equiv a.(a.\mathbf{0} + b.\mathbf{0}) + c.\mathbf{0}$$

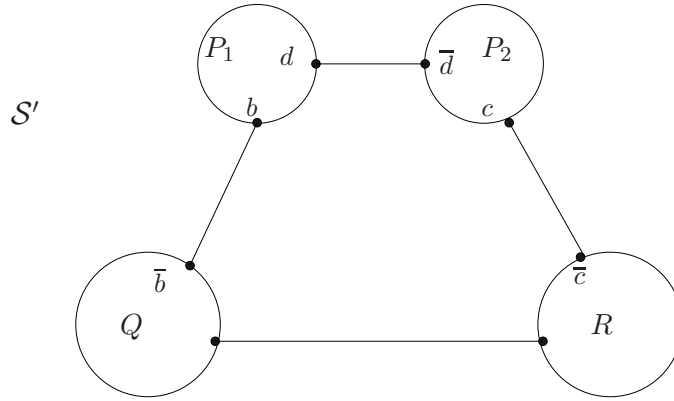
so that $P' \equiv a.\mathbf{0} + b.\mathbf{0}$



3. **Loss of a node:** if $P' : \emptyset$; e.g. $P \equiv a.\mathbf{0}$ so $P' \equiv \mathbf{0}$.



4. **Development of a node:** if P' is a composition of components; e.g. $P \equiv a.P'$ and $P' \equiv (P_1|P_2) \setminus d$.



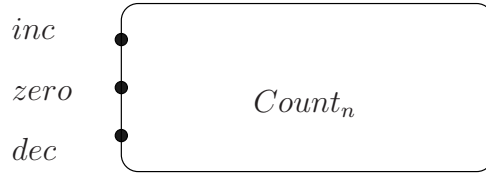
- Some agents' 'size' increases unboundedly.
- The calculus can represent all of these dynamic variations of structure.

Example: A Counter

Problem: We shall design a counter, which can assume any of the natural numbers as its 'states', by increment and decrement actions. The counter should allow a 'test' action if it is zero.

Specification:

$$\begin{aligned} Count_0 &\stackrel{def}{=} inc.Count_1 + zero.Count_0 \\ Count_n &\stackrel{def}{=} inc.Count_{n+1} + dec.Count_{n-1} \quad (n > 0) \end{aligned}$$



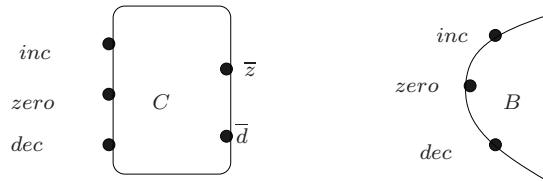
Use of the counter: An agent using the counter and wishing to perform a test-and-decrement action would take the form

$$\overline{zero}.P + \overline{dec}.P'$$

which means that “**if** the counter is zero **then** P **else** decrement the counter and then P' ”.

Implementation

We shall implement the counter by linking together several copies of an agent C and one B , with the following interfaces:



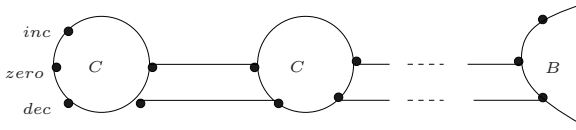
The definitions of C and B :

$$\begin{aligned} C &\stackrel{\text{def}}{=} \text{inc.}(C \frown C) + \text{dec.}D \\ D &\stackrel{\text{def}}{=} \bar{d}.C + \bar{z}.B \\ B &\stackrel{\text{def}}{=} \text{inc.}(C \frown B) + \text{zero}.B \end{aligned}$$

The linking combinator:

$$P \frown Q \stackrel{\text{def}}{=} (P[i'/i, z'/z, d'/d] \parallel Q[i'/\text{inc}, z'/\text{zero}, d'/\text{dec}]) \setminus \{i', z', d, \}$$

Define

$$C^{(n)} \stackrel{\text{def}}{=} \overbrace{C \frown \dots \frown C}^{n \text{ times}} \frown B$$


Verification: To prove the equation

$$C^{(n)} = \text{Count}_n$$

Proof: Consider the family of equations

$$\begin{aligned} X_0 &= \text{inc.}X_1 + \text{zero}.X_0 \\ X_n &= \text{inc.}X_{n+1} + \text{dec.}X_{n-1} \quad (n > 0) \end{aligned}$$

Notice that $\{X_n = \text{Count}_n : n \geq 0\}$ is a solution of this family. Since $C^{(0)} = B$, so

$$C^{(0)} = \text{inc.}C^{(1)} + \text{zero}.C^{(0)}$$

For $n > 0$

$$\begin{aligned} C^{(n)} &= C \frown C^{(n-1)} \\ &= \text{inc.}((C \frown C) \frown C^{(n-1)}) + \text{dec.}(D \frown C^{(n-1)}) \\ &= \text{inc.}C^{(n+1)} + \text{dec.}(D \frown C^{(n-1)}) \end{aligned}$$

It is easy to prove $D \frown C^{(n)} \approx C^{(n)}$. Then by the proposition that if $P \approx Q$ then $\alpha.P = \alpha.Q$, we have

$$\text{dec.}(D \frown C^{(n-1)}) = \text{dec.}C^{(n-1)}$$

Thus $\{X_n = C^{(n)} : n \geq 0\}$ is a solution of the family of equations, where X_n are both guarded and sequential. So we have $\text{Count}_n = C^{(n)}$.

6 Systems with Inductive Structure

- Systems of fixed but arbitrary size.
- The system of size $n + 1$ can be defined in terms of the system of size n .
- The proof of correctness of the systems is by mathematical induction.

A sorting machine

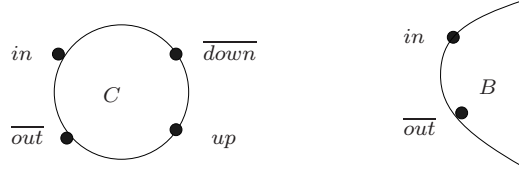
Problem: We would like to build a sorting machine $Sorter_n$, for each $n \geq 0$, capable of sorting n -length sequences of positive integers.

Requirement: $Sorter_n$ has sort $\{in, \overline{out}\}$. It must accept exactly n integers one by one at in ; then it must output them one by one in descending order at \overline{out} , terminated by a zero. After that it must return to its starting state.

Specification: We shall use multisets of integers.

$$\begin{aligned} Sortspec_n &\stackrel{def}{=} in(x_1) \cdots in(x_n).Hold_n(\{x_1, \dots, x_n\}) \\ Hold_n(S) &\stackrel{def}{=} \overline{out}(maxS).Hold(S - \{maxS\}) \quad (S \neq \emptyset) \\ Hold(\emptyset) &\stackrel{def}{=} \overline{out}(0).Sortspec_n \end{aligned}$$

Implementation: We shall build the sorter by linking together n identical cells C and a single barrier cell B :



- C and B are independent of n so that they can be implemented as hardware components of fixed finite sizes, and be used to build sorting machines of any size.
- The idea of designing C is that
 1. It should have storage capacity for two numbers, and be able to compare them.
 2. Its behaviour must have two phases. In the first it receives inputs at in and puts them out at \overline{down} , or change to its second phase, in which it receives inputs at up and (using comparison) outputs them at \overline{out} .

The definition of C is:

$$\begin{aligned} C &\stackrel{def}{=} in(x).C'(x) \\ C'(x) &\stackrel{def}{=} \overline{down}(x).C \\ &\quad + up(y).\overline{out}(max\{x, y\}).C''(min\{x, y\}) \\ C''(x) &\stackrel{def}{=} \text{if } x = 0 \text{ then } \overline{out}(0).C \text{ else } C'(x) \end{aligned}$$

B simply delivers 0 whenever required to do so, thereby triggering a phase-change in its neighbour, which will then perform the same service for its own neighbour, and so on.

$$B \stackrel{def}{=} \overline{out}(0).B$$

Then the sorter is defined as

$$Sorter_n \stackrel{def}{=} \overbrace{C \frown \dots \frown C}^{n \text{ times}} \frown B$$

Verification: To prove the equation

$$Sorter_n = Sortspec_n$$

Outline of the proof: By induction on n .

1. When $n = 0$, we have

$$Sorter_n = B = Sortspec_0$$

2. We have $Sorter_{n+1} = C \frown Sorter_n$.

3. Assume that

$$Sorter_n = Sortspec_n$$

4. We have to show that

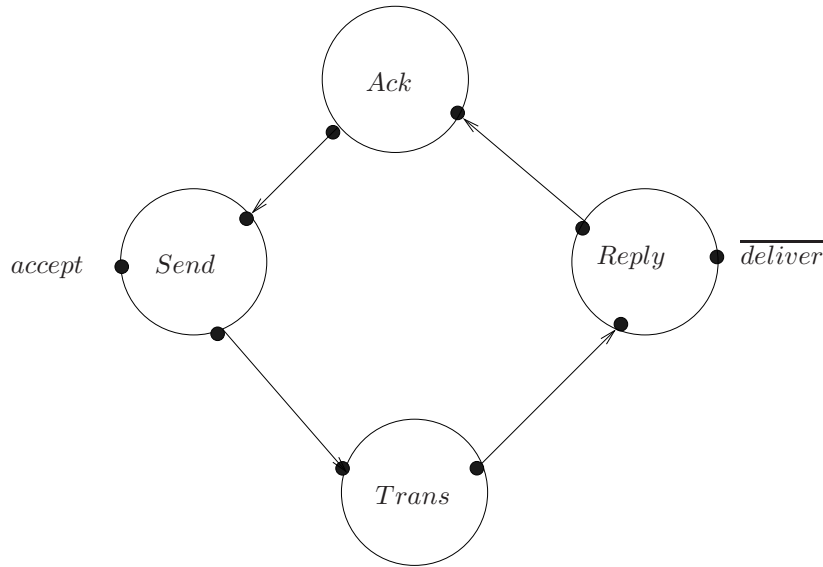
$$Sortspec_{n+1} = C \frown Sortspec_n$$

This is done by using the expansion law many times.

7 The Alternating-bit Protocol

- The design of a concurrent system is often concerned with the design of the communication protocols between its components.
- A protocol is a discipline for transmission of messages from a source to a destination, through transmission mediums.
- Sometimes the medium are unreliable: it may lose, duplicate or corrupt messages.
- Then the protocol should be designed to ensure reliable transmission under such possibly adverse conditions.
- For example, the sender cannot assume that a transmission was successful until an acknowledgement is obtained from the receiver.
- The problem is further compounded if acknowledgements themselves may be lost, duplicated or corrupted.

Problem: Consider a system where the transmission medium consists of communication lines which behaves as unbounded buffers, except that they are unreliable.



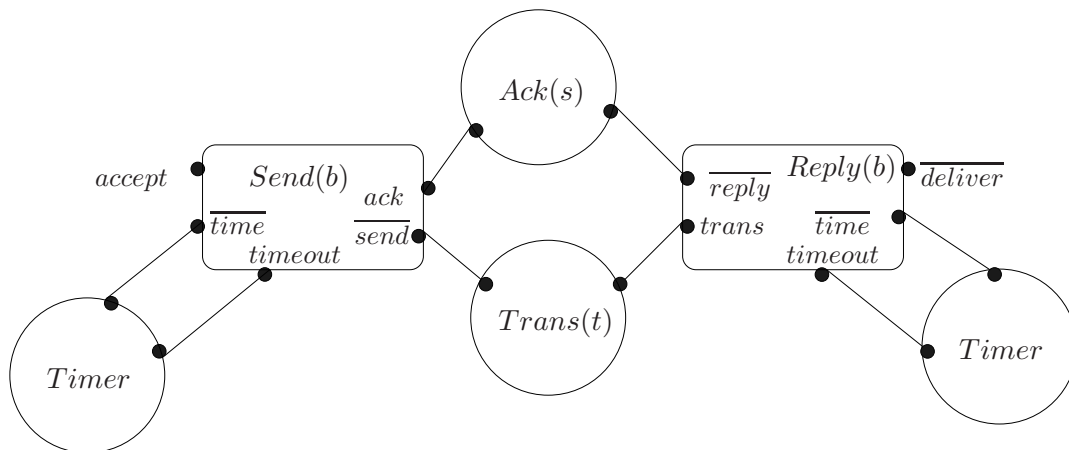
- The source (to the left) and the destination (to the right) are not shown.
- *Trans* and *Ack* are the agents representing the unreliable communication lines.
- *Send* and *Reply* are the agents represent the protocol or discipline.
- How can the Alternating-Bit (*AB*) protocol be specified and verified in our calculus?

Assumption about faulty lines

- They are infinite buffers.
- The *Trans* and *Ack* lines may lose or duplicate.
- They may not corrupt messages.
- They may not change the order in which the messages are sent to them.

Question: How can this assumption about the faults be specified?

Description of the protocol



1. **The behaviour of *Send*:** After accepting a message, it sends it with bit b (0 or 1) along the *Trans* line and sets a timer. It then may
 - get a ‘time-out’ from the timer, upon which it sends the message again with b ;
 - get an acknowledgement b from the *Ack* line, upon which it is ready to accept another message which it will send with bit $\hat{b} = 1 - b$;
 - get an acknowledgement \hat{b} , resulting from a duplication of the previous acknowledgement, which it ignores.
2. **The behaviour of *Reply*:** After delivering a message it acknowledges it with bit b along the *Ack* line and sets a timer. It then may
 - get a ‘time-out’ from the timer, upon which it acknowledges again with b ;
 - get a new message with bit \hat{b} from the *Trans* line, upon which it is ready to deliver the new message (which it will acknowledge with bit \hat{b});
 - get a superfluous transmission of the previous message with bit b , which it ignores.

Requirement specification: We expect the protocol to behave like a perfect buffer of capacity one, i.e. it simply accepts and delivers messages alternately:

$$\begin{aligned} \text{Buff} &\stackrel{\text{def}}{=} \text{accept}.\text{Buff}' \\ \text{Buff}' &\stackrel{\text{def}}{=} \overline{\text{deliver}}.\text{Buff} \end{aligned}$$

The definitions of *Send*, *Reply* and *Timer*

$$\begin{aligned} \text{Send}(b) &\stackrel{\text{def}}{=} \overline{\text{send}_b}.\overline{\text{time}}.\text{Sending}(b) \\ \text{Sending}(b) &\stackrel{\text{def}}{=} \text{timeout}.\text{Send}(b) \\ &\quad + \text{ack}_b.\text{timeout}.\text{Accept}(\hat{b}) \\ &\quad + \text{ack}_{\hat{b}}.\text{Sending}(b) \\ \text{Accept}(b) &\stackrel{\text{def}}{=} \text{accept}.\text{Send}(b) \\ \text{Reply}(b) &\stackrel{\text{def}}{=} \overline{\text{reply}_b}.\overline{\text{time}}.\text{Replying}(b) \\ \text{Replying}(b) &\stackrel{\text{def}}{=} \text{timeout}.\text{Reply}(b) \\ &\quad + \text{trans}_{\hat{b}}.\text{timeout}.\text{Deliver}(\hat{b}) \\ &\quad + \text{trans}_b.\text{Replying}(b) \\ \text{Deliver}(b) &\stackrel{\text{def}}{=} \overline{\text{deliver}}.\text{Reply}(b) \\ \text{Timer} &\stackrel{\text{def}}{=} \text{time}.\overline{\text{timeout}}.\text{Timer} \end{aligned}$$

The definitions of the faulty lines

Informally, we define them by giving all their transitions:

$$\begin{array}{llll} \text{Ack}(bs) \xrightarrow{\overline{\text{ack}_b}} \text{Ack}(s) & \text{Trans}(sb) \xrightarrow{\overline{\text{trans}_b}} \text{Trans}(s) & & \\ \text{Ack}(s) \xrightarrow{\text{reply}_b} \text{Ack}(sb) & \text{Trans}(s) \xrightarrow{\text{send}_b} \text{Trans}(bs) & & \\ \text{Ack}(sbt) \xrightarrow{\tau} \text{Ack}(st) & \text{Trans}(tbs) \xrightarrow{\tau} \text{Trans}(ts) & \text{loss of } b & \\ \text{Ack}(sbt) \xrightarrow{\tau} \text{Ack}(sbbt) & \text{Trans}(tbs) \xrightarrow{\tau} \text{Trans}(tbbs) & \text{duplication of } b & \end{array}$$

Definition of AB

First, we shall compose $Send(b)$ with its $Timer$, and likewise $Reply(b)$ with its $Timer$, since it leads to a simplification:

$$\begin{aligned} Send'(b) &\stackrel{def}{=} (Send(b)|Timer) \setminus \{time, timeout\} \\ Reply'(b) &\stackrel{def}{=} (Reply(b)|Timer) \setminus \{time, timeout\} \end{aligned}$$

Using equational reasoning, we can show that the following hold:

$$\begin{aligned} Send'(b) &= \overline{send_b}.Sending'(b) \\ Sending'(b) &= \tau.Send'(b) + ack_b.Accept'(\hat{b}) + ack_{\hat{b}}.Sending'(b) \\ Accept'(b) &= accept.Send'(b) \\ Reply'(b) &= \overline{reply_b}.Replying'(b) \\ Replying'(b) &= \tau.Reply'(b) + trans_{\hat{b}}.Deliver'(\hat{b}) + trans_b.Replying'(b) \\ Deliver'(b) &= \overline{deliver}.Reply'(b) \end{aligned}$$

So, the system can be written as follows:

$$\begin{aligned} AB &\stackrel{def}{=} (Accept'(\hat{b}) \mid Trans(\varepsilon) \mid Ack(\varepsilon) \mid Reply'(b)) \setminus L \quad \text{where } L \text{ is defined as} \\ L &= \{send, trans, reply, ack\} \end{aligned}$$

For notational reasons we drop all the primes. With such change, AB can also be written, using the above defined notation and not that on the previous page, as

$$AB \stackrel{def}{=} Accept(\hat{b}) \parallel Trans(\varepsilon) \parallel Ack(\varepsilon) \parallel Reply(b)$$

Verification: We shall prove

$$AB \approx Buff$$

Note that since AB and $Buff$ are stable, we obtain $AB = Buff$ as required.

Proof: Find a WB \mathcal{S} which contains the pair $(AB, Buff)$. Here is such a WB:

AB states	$Buff$ states
$Accept(\hat{b}) \parallel Trans(b^n) \parallel Ack(b^p) \parallel \left\{ \begin{array}{l} Reply(b) \\ Replying(b) \end{array} \right.$	$Buff$
$\left. \begin{array}{l} Send(\hat{b}) \\ Sending(\hat{b}) \end{array} \right\} \parallel Trans(\hat{b}^m b^n) \parallel Ack(b^p) \parallel \left\{ \begin{array}{l} Reply(b) \\ Replying(b) \end{array} \right.$	$Buff'$
$\left. \begin{array}{l} Send(\hat{b}) \\ Sending(\hat{b}) \end{array} \right\} \parallel Trans(\hat{b}^m) \parallel Ack(b^p) \parallel Deliver(\hat{b})$	$Buff'$
$\left. \begin{array}{l} Send(\hat{b}) \\ Sending(\hat{b}) \end{array} \right\} \parallel Trans(\hat{b}^m) \parallel Ack(b^p \hat{b}^q) \parallel \left\{ \begin{array}{l} Reply(\hat{b}) \\ Replying(\hat{b}) \end{array} \right.$	$Buff$

\mathcal{S} contains 12 groups of pairs, which arise by choosing either of each pair of the bracketed alternatives. The first group of pairs is

$$(Accept(\hat{b}) \parallel Trans(b^n) \parallel Ack(b^p) \parallel Reply(b), Buff)$$

When we instantiate n and p to zero (0), then we get the pair

$$(Accept(\hat{b}) \parallel Trans(\varepsilon) \parallel Ack(\varepsilon) \parallel Reply(b), Buff)$$

which by the definition of AB is $(AB, Buff)$ as required.

Discussion

- **Divergence:** AB may diverge (or lifelock), i.e. it contains τ -cycles, because of the possibility of indefinite loss and retransmission and because of the possibility of indefinite duplication. But, $Buff$ can never diverge.
- The calculus cannot treat divergence, i.e. \approx and $=$ cannot distinguish the presence and absence of divergences in agents, although it can take care of deadlock.
- In general, the calculus can deal with safety properties of systems, but cannot deal with liveness properties. Temporal logic is good in dealing with liveness properties.
- If the *Ack* and *Trans* lines may only lose or duplicate a finite number of messages consecutively, then AB is convergent.
- The proofs in the calculus often require case analysis. To some extent, case analysis is inescapable in handling realistic systems. But when there are too many cases to analyse, we need to use a help from a proof tool such as for example CWB.
- The advantages of the CCS calculus:
 1. It is simpler than one which takes care of liveness properties.
 2. The bisimulation approach is easy to mechanise, as demonstrated by the CWB.
 3. It is good to start with such a simple theory to understand concurrent systems.