Chapter 1. Modelling Concurrent Systems

Goals for Chapter 1:

- The notion of agents
- The handshake communication between agents
- The five basic agent operators (combinators) for constructing agent expressions
- Understand the meaning of equations between agent expressions
- Examples of simple systems
- Simple examples of specification, implementation and verification process

1 Agents (Processes)

We describe concurrent systems as networks of agents or processes:

- Each agent has its own identity which persists through time.
- An agent is used to identify where an event occurs.
- An agents may be decomposed into a network of several sub-agents acting concurrently and interacting. But the level to which we proceed in decomposition depends upon our present interest, not upon the entities.
- A system itself is an agent.

Example. Leicester University is a network of departments; a department is a network of people:

- A department is an agent with a name, e.g. Maths & Computer Science, which persists through time.
- Departments carry out their activities concurrently with and independently of each other; departments also interacts with each other to collaborate in teaching and research.
- We identify where an activity, such as teaching a course, is carried out by a department.
- A department is again a network of people who are the students and the members of staff in that department; we do not treat a person as a network of parts while we are interested in departments (this is where our decomposition ends), so a person is an atomic agent.
- Leicester University itself is an agent, which can be a sub-agent when we are interested in the network of universities of UK.

Example. A concurrent program P of several processes, say

cobegin $P_1; P_2; P_3$ **coend**

is an agent which is a network of the agents P_1, P_2 and P_3. The names of the processes (agents) persists through the lifetime (i.e. the execution) of the program. These names are used to identify

where a message came from or where a message is sent to. Each process, say P_1, may be a concurrent process itself (i.e. COBEGIN/COEND can be nested). For example, P_1 could be

cobegin $P_{11}; P_{12}; P_{13}; P_{14}$ **coend**

Therefore, the use of the term agent is very broad which means any system whose behaviour consists of discrete actions; an agent which for one purpose we take to be atomic may, for other purposes, be decomposed into sub-agents acting concurrently and interacting.

2 Communication

- Each action of an agent is either an interaction (communication) with other agents, or it occurs independently (may occur concurrently with actions) of other agents.
- Independent actions of an agent are again its own communications with the environment, or communications among the components of that agent.
- The behaviour of a system is its entire capability of communicate (or perform actions).
- The behaviour of a system is exactly what is externally observable.
- To observe a system is exactly to communicate with it.

Communication Media

Problem: Consider the problem of transmission of information from one agent to another. There are many ways to implement transmission of information. Next, we look at several of them.
In general, transmission of information involves three entities; two agents, the *Sender* and *Receiver*, and an entity called the MEDIUM where items of information reside while in transit:

We first assume that items of information are transmitted in the form of messages, i.e.

M1: Each message corresponds to a single act of sending; a message is therefore sent exactly once.

M2: Each act of receiving involves a distinct message; so each message is received at most once.

In what follows, we discuss and classify media according to the discipline under which sending and receiving occur.

1. ETHER

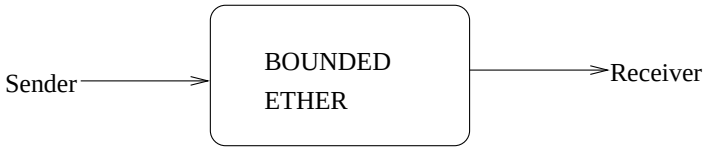
The ETHER protocol is:

(a) The *Sender* may always send a message.

(b) The *Receiver* may always receive a message, provided the medium is not empty.

(c) The orders in which messages are sent and received may be different.

2. BOUNDED ETHER



The BOUNDED ETHER protocol:

(a) The *Sender* may always send a message, provided the medium is not full.

(b) As for ETHER.

(c) As for ETHER.

3. BUFFER

The BUFFER protocol:

(a) As for ETHER

(b) As for ETHER

(c) Receiving oder = Sending order

4. BOUNDED BUFFER

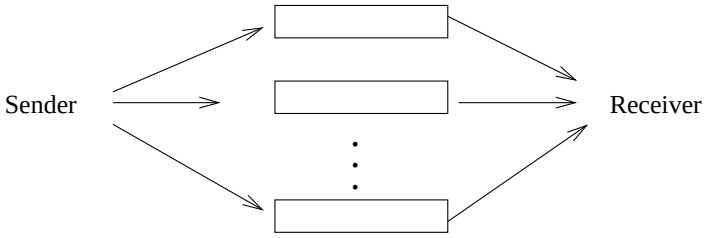
The BUFFER protocol:

(a) As for BOUNDED ETHER

(b) As for ETHER

(c) As for BUFFER

You should by now be reasonably happy about the idea that the transmission of information involves in three objects. However, is it always possible for the items of information in all media to be considered as messages? In other words, do all media always receive and send items of information in the way **M1** and **M2**?
Consider a SHARED MEMORY



The SHARED MEMORY protocol:

1. The *Sender* may always write an item in to a register.
2. The *Receiver* may always read an item from a register.
3. Writing and reading may occur in any order.

This shows that the assumption that items of information should be transmitted in the form of messages is not an essential feature of communication.

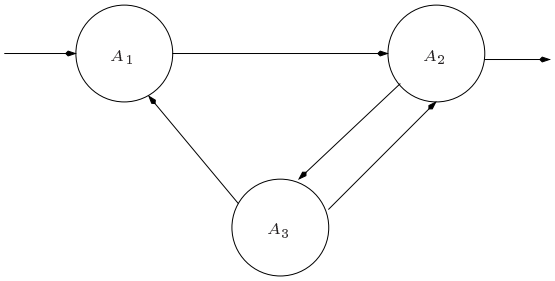
Basic Common Points

- Each arrow is a vehicle for a single action, indivisible in time. Such an action is called an atomic action.
- Each single action consists of the passage of an item between two entities, a sender and a receiver.
- "Performers" and "passive entities" all participate in these single acts of communication.

Decisions

- We take just one kind of agents/performers, which take part in acts of communication
  - "active" entities, *Sender*s and *Receiver*s, and "passive" entities, the media, are all actively taking part in acts of communication,
  - a "passive" medium such as a buffer may be decomposed into agents which communicate with each other to move data around.
- Such an act of communication is experienced simultaneously by both participants.
- These acts of communication are called handshakes.

Graphical Representation



- Linked arrows indicate handshakes between agents.
- Non-linked arrows indicate handshakes with environment.
- We shall treat non-linked and linked arrows quite differently when we consider the behaviour of a system.

3 Agent Expressions

This section deals with the following two questions.

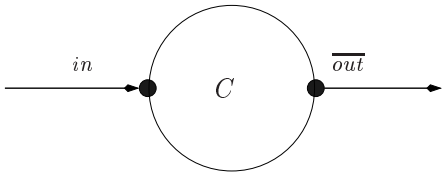
1. How can we formally describe the behaviour of an agent?
2. How can we describe the pattern of communication between agents?

The answers to these two questions form topics of formal specification of and reasoning about concurrent systems. In this section we informally introduce the concepts of formal specification and reasoning by means of examples, and build up our understanding about them. In the next chapter, we will give the formal definitions of the syntax and semantics of these notations.

An agent is represented as an agent expression, which is built up from agent (process) operators (combinators).

- What are the symbols and operators that we need? – informal syntax and semantics
- How to construct expressions for agents? – specification techniques
- What equations should hold between agent expressions? – analysis and reasoning

3.1 A Cell



- An agent has a name, e.g. C
- An agent has some named ports called port labels, e.g. in, $\overline{out}$, through which it communicates with its environment.

Behaviour

The above graphic description does not tell us anything about the behaviour of the agent C. The intended behaviour of C is:

- When empty, C may accept an item at port in;
- When holding a value, C may deliver it at port $\overline{out}$.

Agent Expressions

The behaviour C is described by the following agent expressions:

$$C \stackrel{def}{=} in(x).C'(x)$$

$$C'(x) \stackrel{def}{=} \overline{out}(x).C$$

Remarks

- Agent names, e.g. $C'(x)$, can take parameters.
- The Prefix operators $in(x).$ and $\overline{out}(x)$ represent a half of a handshake at port in and $\overline{out}$ respectively.
- The agent expression $in(x).C'(x)$ represents an agent that performs the handshake $in(x)$ and then proceeds according to the definition of $C'(x)$.
- $\overline{out}(x).C$ outputs the value of x at port $\overline{out}$ and then proceeds according to the definition of C.

Scope of a Variable

A variable is either bound by an input Prefix, for example, $in(x).C'(x)$ or by the left-hand side of a defining equation, for example, $C'(x) \stackrel{def}{=} \overline{out}(x).C$.

Remarks

1. The names of parameters may be consistently renamed. For example, $C'(x) \stackrel{def}{=} \overline{out}(x).C$ can also be written as $C'(y) \stackrel{def}{=} \overline{out}(y).C$.
2. Output Prefix like $\overline{out}(x)$ does not make the variable bound in an expression like $\overline{out}(x).C$. We say that x is free in $\overline{out}(x).C$.
3. Expression $C'(x)$ is auxiliary. We can write the definition of C without the use of $C'(x)$ as follows:

$$C \stackrel{def}{=} in(x).\overline{out}(x).C$$

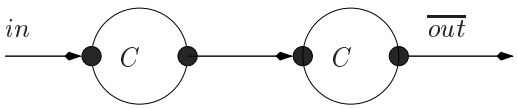
Exercise. Consider the following agent definition (equation).

$$A \stackrel{def}{=} in(x).in(y).\overline{out}(x).\overline{out}(y).A$$

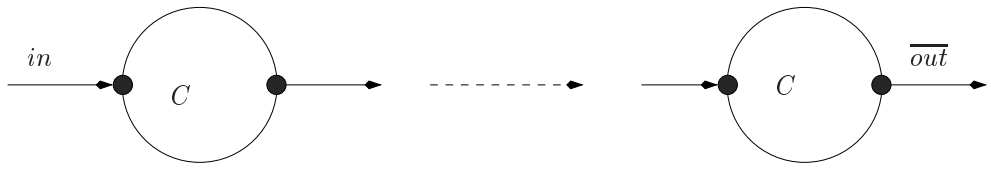
1. How does the behaviour of A differ from that of C?
2. Write the definition of A using a pair of defining equations. Repeat this but with four equations.

3.2 Bounded Buffer

Consider linking two or more copies of C

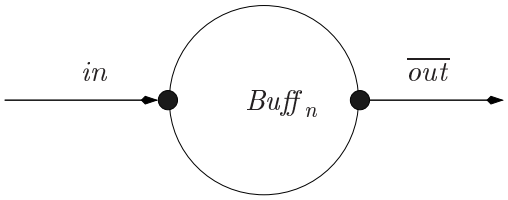


$$C^{(2)} \stackrel{def}{=} C \frown C$$



$C^{(n)} \stackrel{def}{=} C \frown C \frown \ldots \frown C$

C^n behaves as a buffer of capacity n.
A buffer of capacity n



$Buff_n <> \stackrel{def}{=} in(x).Buff_n < x >$

$Buff_n < v_1, \ldots, v_k > \stackrel{def}{=} in(x).Buff_n < x, v_1, \ldots, v_k > + \overline{out}(v_k).Buff_n < v_1, \ldots, v_{k-1} > \quad (0 < k < n)$

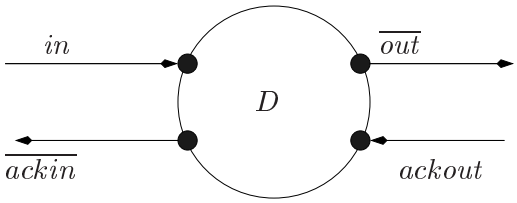
$Buff_n < v_1, \ldots, v_n > \stackrel{def}{=} \overline{out}(v_n).Buff_n < v_1, \ldots, v_{n-1} >$

We have used a new binary agent operator +, called the non-deterministic choice or summation, to construct agent expressions.

- $P + Q$ behaves either like P or like Q; as soon as one performs its first action the other is discarded.
- The binary operator + combines two agent expressions as alternatives, or gives a choice between two agents.
- The Prefix operator 'port-name.' is used as a simple method of producing a sequence of events.
- We can express an agent at different levels of abstraction
  - $Buff_n$ serves as a specification
  - $C^{(n)}$ serves as a design or an implementation.
- We need to have a (equational, algebraic) theory to enable us to prove that the design (implementation) $C^{(n)}$ is correct with respect to the specification $Buff_n$, written as $Buff_n = C^{(n)}$.

More Examples

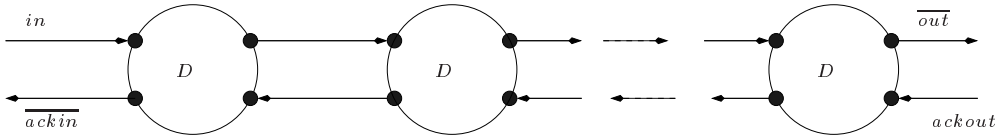
Consider D which is like C, except that the receipt of each value is acknowledged to the sender.



Assume that D acknowledges after the value has arrived at it final destination.

$$D \stackrel{def}{=} in(x).\overline{out}(x).ackout.\overline{ackin}.D$$

Link $n > 1$ copies of D

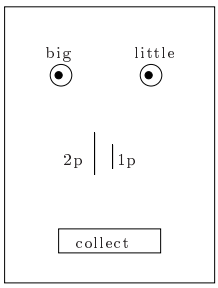


$D^{(n)} \stackrel{def}{=} D \frown \ldots \frown D$. What is the behaviour of $D^{(n)}$?
Assume D' is like D except that it acknowledges its input as soon as it is received

$$D' \stackrel{def}{=} in(x).\overline{ackin}.\overline{out}(x).ackout.D'$$

$D'^{(n)}$ equals $Buff'_n <>$, where $Buff'_n$ differs only slightly from $Buff_n$: it requires the addition of acknowledgement actions.

3.3 A Vending Machine



1. To buy a big chocolate, put a 2p coin, press the button *big*, and collect your chocolate.
2. To buy a little chocolate, put a 1p coin, press the button *little*, and collect.

$$V \stackrel{def}{=} 2p.big.\overline{bigchoc}.V + 1p.little.\overline{littlechoc}.V$$

Imagine a customer $C \stackrel{def}{=} \overline{1p}.\overline{little}.littlechoc.\overline{happy}.\mathbf{0}$, where $\mathbf{0}$ is the deadlocked agent that can perform no actions.

- What will happen when C interacts with V?
- What will happen when the customer pays $1p$ but attempts to presses big?
- What will happen when the customer attempts to pay $5p$?

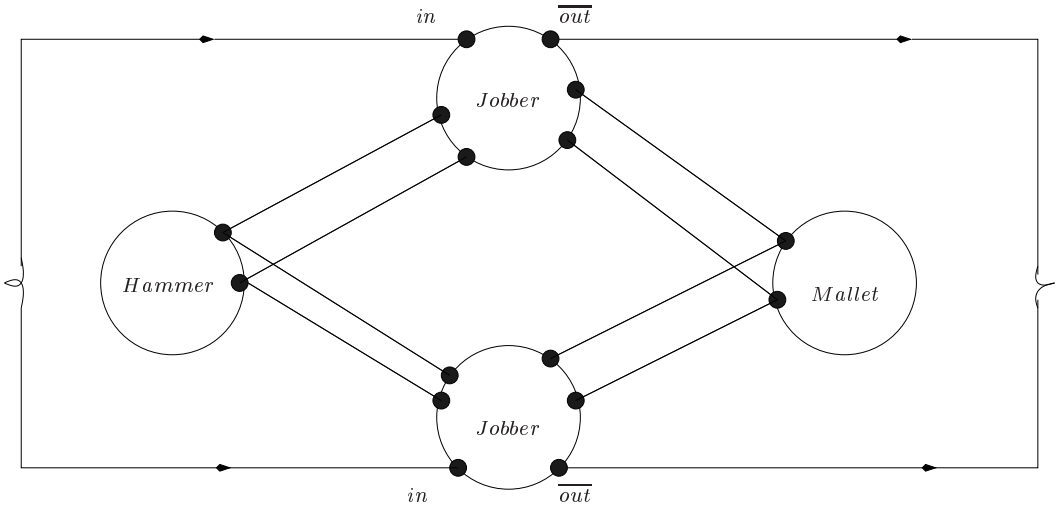
3.4 A Jobshop

- Two *Jobbers* are sharing two tools – *Hammer* and *Mallet* – to make objects.
- Each object is made by driving a peg into a block.
- We call a pair consisting of a peg and a block a *job*.
- Jobs arrive sequentially on a conveyor belt.
- Completed objects depart on a conveyor belt.

What are agents and what are data?

- *Jobbers*, *Hammer* and *Mallet* are the agents.
- *jobs* and objects are the data (values) which enter and leave the system.

A diagram of the Jobshop is as follows.

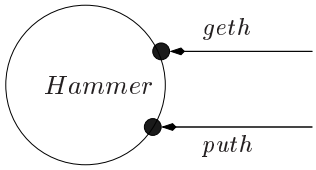


Remarks about the diagram

- A port may be linked to more than one other port.
- This represents the possibility that the two jobbers may compete for the use of a tool.
- This is a potential source of non-determinism.
- in and $\overline{out}$ are the external ports.
- The unlabelled ports (by which the jobbers acquire or release the tools) are internal ones.

Hammer

We regard it as a resource which may just be acquired or released.



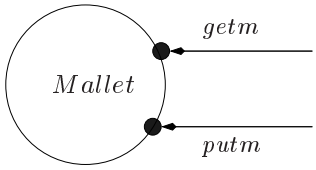
$$Hammer \stackrel{def}{=} geth.Busyhammer$$

$$Busyhammer \stackrel{def}{=} puth.Hammer$$

It is equivalent to

$$Hammer \stackrel{def}{=} geth.puth.Hammer$$

Mallet



$$Mallet \stackrel{def}{=} getm.Busymallet$$

$$Busymallet \stackrel{def}{=} putm.Mallet$$

Remark. Notice, that the behaviour of $Mallet$ is exactly the same as that of $Hammer$ if we replace port names $geth$ and $puth$ with $getm$ and $putm$ respectively.

Sort

- A sort is just a set of port labels.
- An agent P has sort L, if all the (observable) actions of P have the labels in L.
- We write $P : L$ if P has sort L.
- If $P : L$ and $L \subseteq L'$ then $P : L'$.
- We normally find the smallest L such that $P : L$, for example

$$\begin{aligned} Hammer &: \{geth, puth\} \\ Mallet &: \{getm, putm\} \\ Jobshop &: \{in, \overline{out}\} \end{aligned}$$

Jobber

Assume that

- There are easy jobs and hard jobs
  - easy jobs will be done just with hands;
  - hard jobs will be done with hammer;
  - other jobs will be done with either hammer or mallet.
- A function *done* from jobs to objects; $done(job)$ is the object made of job.

The states of $Jobber$:

$Jobber$	Initial state – waiting to receive a job
$Start(job)$	job received
$Usetool(job)$	tool used
$UseH(job)$	hammer used
$UseM(job)$	mallet used
$Finish(job)$	job completed

The $Jobber$ has the sort $\{in, \overline{out}, \overline{geth}, \overline{puth}, \overline{getm}, \overline{putm}\}$. A description of $Jobber$

$$
\begin{aligned}
Jobber &\stackrel{def}{=} in(job).Start(job) \\
Start(job) &\stackrel{def}{=} \textbf{if } easy(job) \textbf{ then } Finish(job) \\
&\quad\ \textbf{else if } hard(job) \textbf{ then } UseH(job) \\
&\quad\ \textbf{else } Usetool(job) \\
Usetool(job) &\stackrel{def}{=} UseH(job) + UseM(job) \\
UseH(job) &\stackrel{def}{=} \overline{geth}.\overline{puth}.Finish(job) \\
UseM(job) &\stackrel{def}{=} \overline{getm}.\overline{putm}.Finish(job) \\
Finish(job) &\stackrel{def}{=} \overline{out}(done(job)).Jobber
\end{aligned}
$$

Composing Agents

The subsystem consisting of $Jobber$ and $Hammer$ is represented by

$$Jobber|Hammer$$

- The binary combinator | is called parallel composition.

- $P|Q$ is a system in which P and Q may proceed independently, but may also interact through complementary ports.
- Two ports are complementary if one is labelled by a label l and the other by $\overline{l}$ for any action (port) label l. For example, $geth$ and $\overline{geth}$, $puth$ and $\overline{puth}$.
- Interaction on complementary ports produces an internal, silent action τ, the action which cannot be communicated upon.
- If $P : L$ and $Q : N$, the $P|Q : L \cup N$, e.g.
  $Jobber|Hammer \ : \ \{in, \overline{out}, \overline{geth}, \overline{getm}, \overline{putm}, \overline{puth}, geth, puth\}$
- Thus, no ports are internalised by composition: e.g. $a.\mathbf{0}|\overline{a}\mathbf{0}$ can perform a, $\overline{a}$ and τ.

Adding another $Jobber$ to the above subsystem produces

$$(Jobber|Hammer)|Jobber$$

The CCS parallel operator (combinator) has the following properties.

- Composition is commutative

  $$Jobber|Hammer = Hammer|Jobber$$

  Recall, that $=$ means 'have the same external behaviour'.
- Composition is associative

  $$(Hammer|Jobber)|Jobber = Hammer|(Jobber|Jobber)$$

  so we can safely omit brackets.
- $\mathbf{0}$ is the identity (unit) for $|$, e.g.
  $$Jobber|\mathbf{0} = Jobber$$
- Unlike $Jobber + Jobber = Jobber$ we **do not** have
  $$Jobber|Jobber = Jobber$$

Hiding Ports—Restriction of Actions

The motivation. Consider Jobshop again.

- More $Jobbers$ can come to the subsystem to share the same $Hammer$.
- To prevent this, we have to hide the ports of $Hammer$.
- In general, a port can be hidden so that no further agents can be connected to it. In other words, hiding a port prevents the system (agent) from communicating (or interacting) with its environment through that port.

To hide a set L of ports of an agent P, we apply the combinator $\backslash L$, called restriction, to P: $P\backslash L$. For example,

$$(Jobber|Jobber|Hammer)\backslash\{geth, puth\}$$

Operator $\backslash L$ hides both the ports L and their complements. Hiding ports must be done with great care, incorrectly hiding a port may cause deadlock or lockout, e.g.

$$(Jobber|Hammer)\backslash\{geth, puth\}|Jobber$$

$$(Jobber|Jobber)|(Hammer\backslash\{geth, puth\})$$

The System *Jobshop*

We compose $Mallet$ with $Jobber|Jobber|Hammer$, hide appropriate ports and obtain

$$Jobshop \stackrel{def}{=} (Jobber|Jobber|Hammer|Mallet)\backslash L$$

where $L = \{geth, puth, getm, putm\}$.

Remarks

- There are many ways of composing a system from given subsystems.
- There are many ways of decomposing a composite systems into subsystems.
- Some of these different ways will be studied as instances of algebraic laws of equivalence.

Relabelling Ports

If we correctly relabel the ports of $Hammer$, we will get $Mallet$.

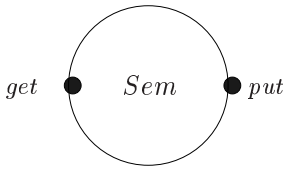
- Extend the "overbar" function to all labels by letting $\overline{\overline{l}} = l$.
- A function f from labels to labels is called a relabelling function if whenever $f(l) = l'$ then $f(\overline{l}) = \overline{l'}$.
- f is written as $l'_1/l_1, \ldots, l'_n/l_n$ if $f(l_i) = l'_i$ and $f(\overline{l_i}) = \overline{l'_i}$ for $i = 1, \ldots, n$, and otherwise $f(l) = l$.
- Given a relabelling function f and an agent P, we use

  $$P[f]$$

  for the agent obtained from P by replacing each port l of P with $f(l)$.

Example

A semaphore



$Sem \stackrel{def}{=} get.put.Sem$

Now, we can redefine $Hammer$ and $Mallet$ using Sem:

$Hammer \stackrel{def}{=} Sem[geth/get, puth/put]$

$Mallet \stackrel{def}{=} Sem[getm/get, putm/put]$

Thus, the Relabelling combinator allows us to give a generic definition for an agent. Such a generic agent can then be re-used as many time as we like by properly renaming its port labels. In other words, this operator allows us to easily make use of agent definitions we already have, where appropriate. The advantage of the feature is that it makes the system development much more efficient, makes the development documentation much shorter and more readable.

Summary of Combinators

So far, we have studied the following CCS operators (combinators):

Name of operator	Syntax
Deadlocked Agent	$\mathbf{0}$
Prefix	$in(x).P$, $\overline{get}.Q$, $a.B$
Summation	$P + Q$
Parallel Composition	$P \mid Q$
Restriction	$P\backslash\{l_1, \ldots, l_n\}$
Relabelling	$P[l'_1/l_1, \ldots, l'_n/l_n]$

When you write CCS expressions, namely expressions constructed with the above operators and agent constants such as, for example, P and Sem, you should use parentheses (brackets) to indicate clearly the arguments for each operator. For example, a parallel composition of $A + B$ and $C + D$ is written as $(A + B)|(C + D)$. The restriction of a in $A + B$ is written as $(A + B)\backslash\{a\}$. The last is not the same as $A + B\backslash\{a\}$, which indicates that a is only restricted in B but not in A. Moreover, $a.(A\backslash L)$ is different from $(a.A)\backslash L$.

Too avoid writing too many brackets we assume that the operators have decreasing binding power in the following order:

- Restriction and Relabelling have the tightest binding, then
- Prefix, then
- Composition, and finally
- Summation.

For example

$$P + a.R \mid b.S\backslash L \qquad \text{means} \qquad P + ((a.R) \mid (b.(S\backslash L)))$$

Important: Not all expressions written with CCS operators are well-formed CCS expressions. For example, $a + b$ is not a valid CCS expression. The following is a list of some of the most common types of simple expressions mistakenly used by students as valid CCS expressions. Can you explain why these are not valid CCS expressions?

- $A + a$, where A is an agent constant and a is an action
- $A.B$, where A and B are agent constants (this does not stand for the composition of agents A and B in sequence!)
- an instance of the last: $(a.\mathbf{0} + b.P).R$
- and another: $(a \mid b).P$
- $a \mid P$
- $P\backslash R$, where both P and R are agents
- $P[R/a]$, where both P and R are agents and a is an action

4 Equality of Agents: a Taster

We write $C^{(n)} = \mathit{Buff}_n <>$ to mean that $C^{(n)}$ and $\mathit{Buff}_n <>$ have the same external behaviour. Other examples are

$$Hammer = Sem[geth/get, puth/put]$$

and the various equivalent expressions of $Jobshop$ in terms of its components.
Consider an n-ary semaphore

$$\begin{array}{lcl} Sem_n(0) & \stackrel{def}{=} & get.Sem(1) \\ Sem_n(k) & \stackrel{def}{=} & get.Sem_n(k+1) + put.Sem_n(k-1) \quad (0 < k < n) \\ Sem_n(n) & \stackrel{def}{=} & put.Sem_n(n-1) \end{array}$$

$$Sem^{(n)} \stackrel{def}{=} \overbrace{Sem|Sem|\ldots|Sem}^{n}$$

We expect that $Sem^{(n)}$ correctly implements Sem_n:

$$Sem_n(0) = Sem^{(n)}$$

Specifications of Jobshop

Let S be defined as follows.

$$\begin{array}{lcl} S & \stackrel{def}{=} & in(job).S' \\ S' & \stackrel{def}{=} & in(job).S'' + \overline{out}(done(job)).S \\ S'' & \stackrel{def}{=} & \overline{out}(done(job)).S' \end{array}$$

We may argue that S is a specification for $Jobshop$:

$$S = Jobshop$$

What other agent you know is similar to S?
Moreover, let

$$Strongjobber \stackrel{def}{=} in(job).\overline{out}(done(job)).Strongjobber$$

We claim that

$$Jobshop = Strongjobber|Strongjobber$$

Thus, $Strongjobber|Strongjobber$ is another specification of the system $Jobshop$.

5 Summary

1. Main ideas:
   - Treat systems as networks of agents.
   - Agents may be atomic, or a composition (a network) of sub-agents, depending on the level of abstraction.
   - An action of an agent is either internal (or silent), independent of other agents, or it is external (or visible) and thus it can be communicated upon by other agents.
   - An invisible action of an agent is nothing more than a communication between sub-agents of that agent.
   - A communication is a handshake between two agents, i.e. an action carried out simultaneously by the two agents.
   - Two agents are equivalent if their external behaviour is the same.
2. Use of mathematics:
   - An agent is represented as an expression in terms of symbols and combinators.
   - Symbols for agent constants and agent variables, symbols for actions (port labels), symbols for values (data).
   - Six combinators (operators): the Deadlocked Agent, Prefix, Summation (Choice), Composition, Restriction, and Relabelling.
3. Two kinds of communication:
   - Synchronisation, as in $Vending\ machine$, $Hammer$, $Mallet$, Sem, etc..
   - Value-passing communication, as in C, $C^{(n)}$, $Buff_n$, $Jobshop$.

6 Exercises

1. Which of the following agents behaves like the cell C?

   (a) $C_1 \stackrel{def}{=} in(x).\overline{out}(y).C_1$

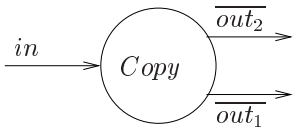
   (b) $C_2 \stackrel{def}{=} in(y).\overline{out}(y).C_2$

   (c) $C_3 \stackrel{def}{=} in(y).out(y).C_3$

   (d) $C_4 \stackrel{def}{=} in(y).C_4'(y)$, $C_4'(x) \stackrel{def}{=} \overline{out}(x).C_4$

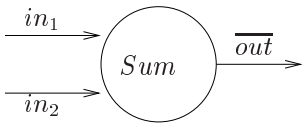
2. Define

   (a) an agent *Copy*:

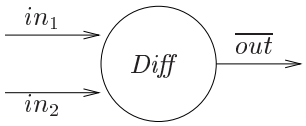
   which inputs a number and outputs it at two ports, repeatedly.

   (b) an agent *Sum*:

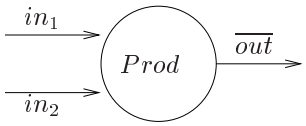
   which inputs two numbers and then outputs their sum, repeatedly.

   (c) an agent *Diff*:

   which inputs two numbers and then outputs their difference, repeatedly.

   (d) an agent *Prod*:

   which inputs two numbers and then outputs their product, repeatedly.

   (e) Build from these agents an agent which repeatedly inputs a pair of numbers at two different ports and outputs the difference of their squares.

   (f) Modify your system if necessary (perhaps adding new components) so that a new pair of numbers cannot be input until the result from the previous pair has been output.

3. Design a user-friendly vending machine VM which can take up to 5p credit, dispense 1p or 2p chocolates and return change, subject to the following constraints.

- VM can either collect coins first, and then accept customer's choice, or vice versa: accept customer's choice first and only then collect coins.
- VM never runs out of chocolates.
- VM does not make a loss or a profit.
- VM cannot hold a credit of more than 5p—this may stop you putting in a coin.
- The chocolate tray of VM cannot hold more than one uncollected chocolate.

4. Describe carefully the behaviour of agent $D^{(n)}$ defined on page 11 of Chapter 1. Define agent $Buff'_n <>$ in the same style as the buffer $Buff_n <>$ from Chapter 1, but with acknowledgement actions added so that $Buff'_n <>$ and $D'^{(n)}$ have the same behaviour, where agents D' and $D'^{(n)}$ are as on page 11 of Chapter 1.

5. Define $C \frown C$ in terms of Relabelling, Composition and Restriction, where C is the agent $Cell$ and $\frown$ is the linking combinator as defined in Chapter 1.

6. The basic characteristics of concurrent systems are listed below.

   - They (may) consist of several parts.
   - Each part may act independently of (or concurrently with) the other parts.
   - The parts may also communicate (or interact) with each other.
   - The behaviour of the system is determined by its entire capability to communicate with the environment.

   Give an account of how the operators (combinators) of CCS can be used (in agent expressions) to capture these characteristics. Illustrate your explanation with examples.

7. Design an agent that describes the production of an omelet. As most of you know, some activities involved in cooking an omelet may be done concurrently. Try to maximise the concurrency involved.

8. Assuming that $\alpha \in Act$ and $E, E_1, E_2 \in \mathcal{E}$, which of the following expressions are not agent expressions, and why?

   (a) $E_1.E_2$
   (b) $\alpha + E_1$
   (c) $\alpha.\mathbf{0} + E_1$
   (d) $(a + b).c.\mathbf{0}$
   (e) $(a.\mathbf{0} + b.\mathbf{0}).c.\mathbf{0}$
   (f) $\alpha.\mathbf{0}.\alpha.\mathbf{0}$
   (g) $\alpha|E$
   (h) $\alpha.(E_1|E_2)$
   (i) $\alpha.E_1|E_2$
   (j) $\alpha.E_1|\mathbf{0}$
   (k) $\alpha.(\mathbf{0}|E)$
   (l) $E\backslash\tau$

(m) $E[a/\tau, b/c]$

(n) $E[\tau/a, b/c]$

9. **Counter**. An object starts on the ground, and may move up. At any time thereafter it may move up or down, except that when on the ground it cannot move any further down. But when it is on the ground, it may move around. Write a specification of this object.

10. **Change giving machine**. A machine with the sort $\{in10p, \overline{out5p}, \overline{out2p}, \overline{ou1p}\}$ (repeatedly) gives change for 10 pence. After inserting 10p, the customer may choose any combination or sequence of 5p, 2p and 1p coins, provided the total value equals 10 pence. Construct the agent CH to behave as described above. Draw the transition graph for CH.