COM3190: Theory of Distributed Systems

Andrew Hughes

COM3190: Theory of Distributed Systems

by Andrew Hughes

Autumn Semester, 2003-2004 Edition

Table of Contents

List of Tables

Preface

The material for these notes was accumulated via lectures given by Dr. Mike Stannett at the University of Sheffield, during the Autumn semester of 2003-2004. The official university website for this module is available at: http://www.dcs.shef.ac.uk/~mps/courses/com3190/ (http://www.dcs.shef.ac.uk/~mps/courses/com3190/)

The recommended texts are:

- Communicating With Mobile Systems with the π Calculus (1999) - Robin Milner
- The π Calculus - A Theory of Mobile Processes. (2001) - Davide Sangiorgi and David Walker.
- CCS: Communication and Concurrency (1989) - Robin Milner

This page was last updated on: 24/02/2004

Chapter 1. Tutorial 1 - 30th of September, 2003, 3pm

Regular models, such as Finite State Automata (FSA), PushDown Automata (PDA) and Turing machines only work sequentially.

1.1. Recommended Texts

The first of the recommended texts can be quite hard to read. The second focuses on the mathematical bits of the course. The third is easier to read than Robin Milner's other book, but tends to be very expensive. The π calculus differs from CCS in that it deals with mobility. The course will present a more understandable version of the first book. In addition, Robin Milner's tutorial notes are much easier to understand.

1.2. The Job Shop

Two workers are working in a factory shed. They construct completely pointless objects from two pieces of wood: one is rectangular, the other is a cylinder which fits in to a hole in the other. The workers have two tools at their disposal: a soft wooden mallet and a hammer. For an easy job, either tool can be used. For a hard job, only the hammer can be used. It can be assumed that each job is labelled as easy or hard.

There are interaction problems involved in this task; a tool needed for a job may already be in use for example. What happens for each job also depends on previous jobs.

Chapter 2. Lecture 1 - 3rd of October, 2003, 10pm

Central to this subject is an understanding of certain terms. In a ’concurrent’ system, more than one thing can be happening at the same time. Conversely, a ’sequential’ system only ever has one thing happening at one time (operations are performed sequentially). In a ’distributed’ system, things can happen in different places, while in a ’monolithic’ system, everything happens in one place. The terms are fairly interchangable; when several things happen at the same time, they usually happen in different places.

Behaviour is the thing a machine computes, |F|. Finite automata stop at a final state. Pushdown automata stop at a final state where the stack is empty. Turing machines stop at the end of their program, with the output of the tape as the result.

2.1. Church/Turing Thesis

The Church/Turing thesis states that anything you can compute ’effectively’ can be computed on a Turing machine. The important word here is ’effectively’. This is a 1930’s term, which refers to anything a human being can do without using intuition, like when working out a proof.

For example, given a random statement, such as ’If n is even, find n / 2. If n is odd, find 3n + 1’, can it be shown to be a theorem?

2.2. X Machines

In an x-machine, memory is manipulated on each transition. The labels on the transitions are the names of functions. The X in X-machines stands for the unknown thing that is manipulated.

2.2.1. An X Machine for a drinks machine

1. Think of the basic functions: Insert Coin_{10}, Insert Coin_{20}, Press Button, Remove $\text{Drink}_{\text{Tea}}$, Remove $\text{Drink}_{\text{Coffee}}$
2. What is the memory in the system? Amount of change, supply of drinks.
3. X = CASH $\times$ DRINKS

Insert Coin_{10}: (Cash, Drink) $\longrightarrow$ (Cash + 10, drink)

Insert Coin_{20}: (Cash, Drink) $\longrightarrow$ (Cash + 20, drink)

Remove $\text{Drink}_{\text{Tea}}$: (Cash - 40, Drink - 1 item) if cash $\geq$ 40, (Cash, Drink) otherwise.

X-machines aren’t as obvious as they look. They have a huge language: $(\text{Insert Coin}_{10}|\text{Insert Coin}_{20}|\text{ Remove Drink})^{*}$, but many combinations are pointless as remove drink won’t work without the required amount of cash.

X-machines split apart the control structure and the data. Given a function f:n $\longrightarrow$ n + 1 on a line of an assumed length of 1, a certain amount could have been added at a certain point along the line (e.g. ½ at the midpoint). This trivial extension allows for continuous function generation, as infinite divisions of the line can theoretically be made.

2.3. Groups

Operations on the real world are represented by groups e.g. $ab \subseteq G \longrightarrow ab^{-1} \subseteq G$.

Chapter 3. Lecture 2 - 3rd of October, 2003, 11pm

3.1. Return To The Jobshop

3.1.1. Agents

Table 3-1. Job Shop Agents

Yes	**No**
Jobbers	Table
Tools	Components

CCS doesn’t have functions; it is based on communication sharing and interaction. Agents have an effect on what goes on around them. The hammer prevents itself being used if it is already in use, making it an agent.

3.1.2. CCS Model of the Job Shop

On a CCS diagram, the nodes are communication ports and the lines are communication channels. The hammer has two input ports: geth and puth. The mallet has two similiar input ports: getm and putm. When the hammer recieves communication from the geth port, it becomes a BusyHammer. Communication from the puth port turns it back into a Hammer. The same occurs with the mallet: getm makes it a BusyMallet, and putm makes it a Mallet again.

Hammer = geth.BusyHammer

BusyHammer = puth.Hammer

Mallet = getm.BusyMallet

BusyMallet = putm.Mallet

A jobber has one input port, called simply ’in’, which recieves a job from the outside world. A jobber also has five output ports: (these are denoted in CCS with a line over the top) geth, puth, getm, putm and out, for output of a job to the outside world.

Jobber = in(job).BusyJobber

BusyJobber = geth.BusyJobberWithHammer

BusyJobber = getm.BusyJobberWithMallet

BusyJobberWithHammer = puth.out(job).Jobber

BusyJobberWithMallet = putm.out(job).Jobber

The job shop is represented as a whole by:

JobShop = (Jobber|Jobber|Hammer|Mallet)

3.1.3. Representing an agent as a state machine

Individual agents can be represented as a state machine, with the ports as the transitions. For example, the Jobber starts in the Jobber state and moves to the BusyJobber state through the transition in(job). A state machine can be used to represent the same information as the set of equations above for an agent. In a simpler fashion, the hammer moves from Hammer to BusyHammer on the geth transition and from BusyHammer to Hammer on the puth transition.

3.1.4. Synchronization

The hammer only does anything when the jobber communicates with it. Outputs and inputs have to be synchronized. P and Q running in parallel and taking matching inputs and outputs can become P′ and Q′ running in parallel if synchronization occurs. Synchronization is represented by τ and is an invisible process, internal to the system. For example, if BusyJobber can go to BusyJobberWithHammer on geth, and Hammer goes to BusyHammer on geth, this implies that BusyJobber and Hammer running in parallel can become BusyJobberWithHammer and BusyHammer running in parallel via the τ transition.

Chapter 4. Tutorial 2 - 7th of October, 2003, 3pm

4.1. Tutorial Sheet 1

1. One possible solution is to make each state represent a certain amount of money being inserted. When a 10p or 20p piece is entered, a transition occurs to a state with a higher monetary value. When tea or coffee is request, a transition occurs to a state with an appropriate lower monetary value. This solution, however, requires an infinite number of states, as any amount of money could be entered before tea or coffee is requested. An alternative method is to have the 40p and 50p states looping on the addition of extra money, and the returning transitions for tea and coffee returning an unspecified amount of change. There are obvious problems with representing the drinks machine as a simple finite state machine.
2. A finite state machine's behaviour is the set of strings generated. In this example, both machines recognise the same language of {a, ab}. However, one machine is non-deterministic while the other is deterministic. In the first machine, it is possible that only 'a' would be recognisable if the wrong path was chosen. Thus, the definition of behaviour for finite state machines is not good enough.
3. There are several operational differences between these two functions. For example, the greater number of operations in one than the other means that it would usually take longer to process. Function 1 also contains more assignments, and in function 2, it is possible that x could be a constant. However, behaviourally, the difference between these functions is due to the fact that there is more chance for the value of x to be changed inbetween operations in function 1 than in function 2. If another program, Prog, is run in parallel with function 1 and function 2, then function 1 may not result in 10 while function 2 will (for example, if Prog is another copy of function 2, then the result of function 1 may become 20). Thus, the functions are behaviourally different.

4.2. Another Example

Imagine two machines, both with two buttons A and B, which are enabled by an internal process, τ. The first machine responds to presses of buttons A or B when enabled, but the second only responds to one or the other (which one depends on the internal process). Are these machines different? The first machine could be said to provide more choice, but the user is unaware of this until they press one of the buttons (if they press A, they don't know whether or not B would have worked). The difference between the machines is not what they can do, but what they can refuse to do. The second machine can refuse to interact upon the press of the button which is currently inoperable, and thus differs from the first machine, where both buttons always work.

4.3. Specifications

Specifications remove ambiguity and provide legal protection; a client unhappy with a product has no grounds for legal action if the product fits the specification to which they agreed. However, as can be seen in the above examples, different specifications can be produced for a particular system via different interpretations of what is required.

Chapter 5. Lecture 3 - 10th of October, 2003, 10am

5.1. CCS Equations

Examples:

- `SleepingPerson = wakeUp.getUp.AwakePerson`
  `+ wakeUp.goBackToSleep.SleepingPerson`
- `DrivingCar = brake.StoppedDriver`

The left hand side can be replaced by the right-hand side, as they are equivalent i.e. brake.StoppedDriver $\longrightarrow^{\text{brake}}$ StoppedDriver. The rule is a.P $\longrightarrow^{a}$ P.

P = Q + R where P = SleepingPerson, Q = wakeUp.getUp.AwakePerson and R = wakeUp.goBackToSleep.SleepingPerson. Q $\longrightarrow^{\text{wakeUp}}$ Q' where Q' = getUp.AwakePerson and R $\longrightarrow^{\text{wakeUp}}$ R' where R' = goBackToSleep.SleepingPerson. So what happens from Q + R? We have a choice of taking the path from Q to Q' or from R to R'. Whichever choice we make, we are then limited to the choices at this new action (either Q' or R'). Thus, if Q $\longrightarrow^{\text{wakeUp}}$ Q', this implies that Q+R $\longrightarrow^{\text{wakeUp}}$ Q'. Likewise, if R $\longrightarrow^{\text{wakeUp}}$ R', this implies that Q+R $\longrightarrow$wakeUp R'. In both cases, we don't end up with Q'+R'. This implies a state where two possible actions can be taken: we can go to SleepingPerson on goBackToSleep or to AwakePerson on getUp. If we draw a diagram to illustrate the transitions from Q + R, it is clear that neither transition from this state takes us to such a state. Both transitions are wakeUp, but one takes us to Q' and one to R'. Once we are at Q', we have chosen a path which leads to an AwakePerson, and we can no longer take the alternate path (all the alternatives are discarded). A process is simply a load of actions, one after the other.

5.2. Synchronization

For two processes, Q and R running in parallet, there are 3 choices: either Q executes an action with another process other than R, R executes an action with another process other than Q or they synchronize with each other. If Q = a.Q' and R = *a*.R' (actions in italics are outputs), then the following possibilities can occur:

- Q interacts with a process other than R, which results in Q' running in parallel with R. More formally, if Q $\longrightarrow^{a}$ Q', this implies that Q|R (Q and R running in parallel) $\longrightarrow^{a}$ Q'|R.
- R interacts with a process other than Q, which results in Q running in parallel with R'. More formally, if R $\longrightarrow^{a}$ R', this implies that Q|R (Q and R running in parallel) $\longrightarrow^{a}$ Q|R'.
- Q and R synchronize, resulting in the τ action and Q' and R' running in parallel. More formally, if Q $\longrightarrow^{a}$ Q' and R $\longrightarrow^{a}$ R', this implies that P|Q $\longrightarrow'^{\tau}$ Q'|R'.

Q and R have a shared communication channel, a with Q having the input and R having the output. Therefore, they have a third choice of communication using the invisible internal channel, τ.

5.2.1. The Buffer Example

Imagine a situation with three actors: Sender, Reciever and Buff. Tranmissions occur between the sender and receiver via Buff. Each transmission must be acknowledged. Sender and Buff share two communication channels, in and ackSent. Sender outputs on *in* and recieves input on ackSent. Buff recieves input on *in* and outputs on *ackSent*. Buff also shares two communication channels, out and ackRecv, with Receiver. Buff outputs on *out* and recieves input on ackRecv. Receiver outputs on *ackRecv* and recieves input on out.

Sender = *in*(x).ackSent.Sender

Receiver = out(x).*ackRecv*>.Reciever

Buffer = in(x).*ackSent*.*out*(x).ackRecv.Buffer

The values (the x's) can be left blank. A flow diagram shows where these communication channels are. A transition diagram tracks the state of an agent or of agents running in parallel. A transition diagrm for the Sender wouuld show the movement of the Sender between its two states of Sender and ackSent.Sender via the transitions *in*(x) and ackSent. Likewise, tthe Receiver transitions bbetween two states, Receiver and *ackRecv*.Receiver via the transitions out(x) and *ackRecv*. The Buff transition diagram is more complicated, using three states, Buff, *ackSent*.*out*(x).ackRecv and *out*(x).ackRecv and four transitions, in(x), *ackSent*, *out*)(x) and ackRecv.

Chapter 6. Lecture 4 - 10th of October, 2003, 11pm

6.1. More On The Buffer Example

When Sender and Buff are running in parallel, either the Sender (the left side of a transition diagram) does something, the Buff (the right side of a transition diagram) does something or they both do something together (they synchronize). From Sender and Buff running in parallel, there are three different paths to take. We can either end up with ackSent.Sender and Buff running in parallel, Sender and *ackSent*.*out*(x).ackRecv running in parallel or ackSent.Sender and *ackSent*.*out*(x).ackRecv running in parallel. A transition for agents running in parallel can be used to represent this. From each of these three new states, there are another possible three states. However, from our point of view, some of these are useless. We don't want the Sender to send data anywhere but the Buff, and we don't want the Buff to receive data from anywhere but the Sender. There is only one way in CCS to tell something that it can't synchronize with something else -- this is called hiding.

6.2. Hiding

Let's take another example. Imagine that an arrangement has been set up whereby, when I turn the light off and go out, the neighbour comes round to feed my cat.

Me = *light*.meOut

Neighbour = light.feedCat

In this example, what we want to happen is for the two processes to synchronize. When the light goes out, the neighbour should come and feed the cat. However, two other possibilities exist: Me may synchronize with something else when outputting on the light communication channel, and thus the neighbour won't come and feed the cat. The Neighbour may also synchronize with something else and go and feed someone else's cat. If we introduce another process, we can see how this becomes a more serious problem.

Burgular = light.robsHouse

When the Burgular process recieves the light trigger, he will come and rob the house. Currently, there is a possibility that Me will synchronize with this process instead of the Neighbour process. We need the outside world (the Burgular process) to not be able to see the light communication channel. Hiding the light prevents the neighbour feeding someone else's cat and the burgular robbing the house. Only the internal synchronization between Me and the Neighbour can take place. The notation for this is:

(Me|Neighbour) \ {light}

or more generally:

P \ {hidden actions}

would hide the actions between { and }. Hidden actions are internal to the system and can only be used for synchronization.

6.3. Deadlocks

Systems running in parallel can deadlock. This occurs when each process is waiting for the other to do something. This can only occur in concurrent systems.

Five philosophers are sat around a table. Each has a single fork. A bowl of spaghetti is placed in the center of the table. Two forks are required to eat from the bowl. The system will deadlock if each philosopher picks up their fork at the same time (as everyone is waiting for another fork). One of the philosophers could also be blocked from ever eating the spaghetti.

An implementation must meet the constraints of a specification. For example, if the specification requires a ten minute running time, the implementation must complete in ten minutes or less.

6.4. Simulation

An LTS is a labelled transition system. If (Q,T) is a LTS, Q is the set of states and T the set of transition relations (Q x T x Q). A binary relation, S over Q, is a subset of Q x Q relations ($S \subseteq Q \times Q$). If P simulates Q, it can do everything Q can do.

The 0 process doesn't do anything, and is equivalent to a stop. Given a process, P, where P = a.0 and a process, Q, where Q = a.0 + b.0, do they simulate each other? Q can do everything P can do (a.0), so it simulates it. More formally, if a transition $R \longrightarrow^{a} R'$ exits, and R is related to S, then the transition $S \longrightarrow^{a} S'$ exists. Processes can be similiar, but not equivalent. In the above example, Q simulates P (this can be seen easily) and P simulates Q (each transition in Q can be linked to one in P), but they are not equivalent processes.

Chapter 7. Tutorial 3 - 14th of October, 2003, 3pm

7.1. Tutorial Sheet 2

1. A = *a*.0

   B = a.0

   The processes, A and B, running in parallel can evolve to 0 and B running in parallel, A and 0 running in parallel or 0 and 0 running in parallel. Both A and 0 and 0 and B can evolve further to 0 and 0 running in parallel.

   If P = A|A|B|B, then this represents two A processes and two B processes all running parallel. A flow diagram would show the communication channels between these processes (four, all labelled a), while a transition diagram would show the possible paths that can be taken from A|A|B|B.

   In the flow diagram, each A process links with both B processes via a communication channel, a. In the transition diagram, A|A|B|B can evolve to A|B|B, A|A|B or A|B (the 0 process can be ignored). A|B|B can evolve to A|B, B|B or B. A|A|B can evolve to A|B, A|A or A. A|B can evolve to A, B or 0. A and B both evolve to 0.

   CCS only allows one thing at a time (so both pairs can't synch together at the same time). However, this difference is unintelligable from the outside.

2. L = words.*nod*.T

   T = *words*.nod.L

   Gossip = L|T

   There are three possibilites for gossip: the Listener can listen to someone other than the Talker, the Talker can talk to someone other than the Listener or they can synchronize with each other.

   The flow diagram shows the two processes, Listener and Talker, and the two communcation channels, words and nod. The transition diagram shows how the processes evolve when running in parallel. L|T evolves to either nod.T|T, L|nod.L or nod.T|nod.L. nod.T|T evolves to T|T, nod.T|nod.L. L|nod.L evolves to nod.T|nod.L or L|L. nod.T|nod.L evolves to T|nod.L, nod.T|L or T|L. T|T evolves to T|nod.L. L|L evolves to L|nod.T. nod.T|L evolves to T|L or nod.T|nod.L. nod.L|T evolves to T|L or nod.L|nod.T.

3. P simulates Q as it contains Q. Representing the processes as points enables the paths between processes to be clearly seen. PSQ means that Q simulates P. Therefore, if P $\longrightarrow^{a}$ P', Q $\longrightarrow^{a}$ Q'. If P is known to do something, we need to show that Q does it too. The processes link on which actions can be performed. If a process can't perform any actions, any process can simulate it. Each process maps with something with the

same capabilities. In this example, there is simulation in each direction, but they are not bisimiliar as the same linking doesn’t work for both.

Chapter 8. Lecture 5 - 17th of October, 2003, 10am

8.1. Simulation

$Q =^{def} a.(b.c.0 + b.d.0)$

$R =^{def} a.b.(c.0 + d.o)$

Both Q and R are sequential processes, and include no parallel components. 0 is the single terminal state and should be shown as a single process on a diagram, shared by all routes. Processes don't need to be named on diagrams - this is implicit.

R simulates Q if it can do everything Q can. Q can make any move. R has to be able to make the same move or lose the game. With the classic example, P = a.(0 + b.0) and Q = a.b.0, Q can never win when P simulates Q and P can never win when Q simulates P. It is obvious that P and Q are not equivalent, however (P allows a to be chosen, and then prevents any other route being chosen), so we need a better way of defining eqiuvalence.

Every process can simulate 0. For each point on the graph, each process must simulate the other. If Q simulates P, it can always match P's moves. If P takes the a to 0, then Q can also perform an a, taking it to b.0. Q's b.0 can simulate 0, so this path can be simulated. Similarly, if P takes the a to b.0, Q can also take an a to b.0. From here (b.0, b.0), P can only take a b. Q can match this move, leading to (0, 0) which can also be simulated to Q. Therefore, Q can simulate P.

8.1.1. Bisimulation

Bisimulation is where the same simulation works in both directions. P must simulate Q and Q must simulate P *with the same simulation* in both directions (P ~ Q are bisimiliar). S^{-1} is the inverse of the simulation (the arrows are reversed), so for P and Q to be bisimiliar, PSQ and $QS^{-1}P$ must exist (or vice versa).

8.1.2. Relations

Simulations occur when there is a relationship between the processes in P and Q. A relation is like a function, but more than one x maps to each y (e.g. $y = x_2$)

8.2. Hiding

Take a.P|*a*.Q. This can evolve in three ways. a.P can perform the a action with a process other than Q, so that we end up with P|*a*.Q. From here, Q can perform the action *a* with a process other than P, resulting in P|Q. Similarly, a.P|*a*.Q can evolve to a.P|Q via Q performing the *a* action with a process other than Q. This can then evolve to P|Q when P performs the a action with another process. The third possibility is that the processes synchronize

and we go directly to P|Q. Hiding a means that the two side branches disappear; we can only go directly to P|Q. When the hiding of a is referred to, this includes both input and output transitions.

8.3. CSS Equalities

CSS defines certain things to be equal:

- P|Q = Q|P
- P + Q = Q + P (obvious diagramatically)
- P|0 = P (the 0 process is ignored)
- P + 0 = P (appears the same to the outside world)
- P|(Q|R) = (P|Q)|R
- P + (Q + R) = (P + Q) + R

CCS is an example of process algebra. Process algebra turns concurrency (and mobility, in the case of the π calculus) into equations.

Chapter 9. Lecture 6 - 17th of October, 2003, 11am

9.1. Tutorial 2, Question 3

This is a buffer that can hold one piece of data:

$Buff_{in,out}$ = in(x).*out*(x).$Buff_{in,out}$

This is a buffer that can hold two pieces of data:

$Buff_{in,mid}$ = in(x).*mid*(x).$Buff_{in,mid}$

$Buff_{mid,out}$ = mid(x).*out*(x).$Buff_{mid,out}$

Hiding the mid communication channel means that only in and out are visible to the outside world. The two Buff processes are forced to synchronize over the mid channel. The buffer holds a maximum of two values, which come out in the same order they went in. The values need to be tracked, leading to an infinite diagram.

With mid unhidden, two transitions are allowed from the start position of $Buff_{in,mid}$|$Buff_{mid,out}$: *mid*(a).$Buff_{in,mid}$ | $Buff_{mid,out}$ on in(a) and $Buff_{in,mid}$ | out(b).$Buff_{mid,out}$ on mid(b). a and b are different values. In this case, $Buff_{mid,out}$ receives b from a process other than $Buff_{in,mid}$. From *mid*(a).$Buff_{in,mid}$ | $Buff_{mid,out}$, the processes can synchronize. This also leads to $Buff_{in,mid}$ | out(b).$Buff_{mid,out}$, but in this case b = a. Alternatively, *mid*(a).$Buff_{in,mid}$ can synchronize with a process other than $Buff_{mid,out}$, leading back to $Buff_{in,mid}$|$Buff_{mid,out}$. The final possibility is that $Buff_{mid,out}$ synchronizes with another process, leading to *mid*(a).$Buff_{in,mid}$ | out(b).$Buff_{mid,out}$ on mid(b). We can also reach here from $Buff_{in,mid}$ | out(b).$Buff_{mid,out}$ with in(a). mid(a) also operates between these two states in the opposite direction. Finally, as with the other initial state, $Buff_{in,mid}$ | out(b).$Buff_{mid,out}$ can return to the start with out(b).

Hiding the internal mid channel simplifies this dramatically. As no other process can use the mid channel, there is only one route from each state. From the start, we must take in(a) in order to put the processes in a state where they can synchronize. Following synchronization, the out(b) transition leads back to the start. However, this simple idea ignores the possibility of another value being input before the previous value is output. If this occurs, we end up with the state *mid(b)*.$Buff_{in,mid}$ | out(a).$Buff_{mid,out}$. The result is an infinite diagram, as we have to allow for each new value (a, b, c, etc.). The only way to avoid this would be to hypotheise that a is set to b, when a is finally output, thus allowing us to return to the state when a is in the buffer. More generally, we need to reuse previous values to avoid an infinite diagram.

9.2. The λ Calculus

$\{(x,y) \mid y = x^2\}$ is an infinite object, as there are infinitely many pairs. The λ caclulus is one method (the other being Turing machines) proposed to model the computation of theorems. The λ calculus sees functions as instructions, and creates the idea of functions of functions. The previous example could be represented as $\lambda x.x^2$ where λ represents 'for all functions'. The λ calculus thus allows functions to be made computable by representing them in terms of other functions. For example, $\lambda x.fx$ returns another function.

9.3. Summing Up CCS

Milner realised that hidden channels work like local variable names. Thus, communication structures mimic data structures.

Glossary

CCS

: Calculus of Communicating Systems

Chapter 10. Tutorial 4 - 21st of October, 2003, 3pm

10.1. Sheet 3

1.

   a. P = a.b.0 + b.a.0, Q = a.0|b.0

      P can take an a to b.0 and then a b to 0, or it can take a b to a.0 followed by an a to 0. For Q to simulate P, it must be able to perform the same moves. From a.0|b.0, assuming that a and b aren't matching input and output ports, two routes can be taken: an a to 0|b.0 or a b to a.0|0. If P takes an a, then Q can also take an a. This leads P to the state b.0 and Q to 0|b.0. Under CCS, b.0|0 = b.0, so Q simulates P for this route. If P instead takes a b from the start, it leads to a.0. Similiarly, Q leads to a.0|0 which is equal to a.0. Therefore, Q simulates P. P can also simulate Q by the same route, making the processes bisimiliar. Simple diagrams of the two processes shows the identical structure, barring process names, clearly. As each parallel process in Q evolves separately, it is effectively the same as an or operation. Either a.0 or b.0 will evolve first, and, once this has happened, the only possibility is for the other process to evolve (as the first process becomes 0).

   b. Allowing the processes in Q to synchronize prevents P from simulating Q, as it has no way of performing the τ operation.

2.

3.

   a. For X = x.0, the flow diagram consists of simply the process X with one port x and no communication channels. The transition diagram should show the process X going to the process 0 on an x transition.

   b. Same as the above.

   c. For Z = X + Y, there is no obvious way to draw the diagram in CCS. The normal way is to draw the process Z with two ports, x and y. Alternatively, the inner process X and Y can be shown inside the Z process with their appropriate ports. The transition diagram shows the Z process going to 0 on either x or y.

   d. The easiest way to draw P = (a.X|*a*.X) + (a.Y|*a*.Y)'s transition diagram is to draw both sides of the + operator separately first. The two start processes are then amalgamated into one P process, with six transitions (two as, two *a*s and two τs). These lead to X|*a*.X, Y|*y*.Y, a.X|x, a.Y|y, X|X and Y|Y respectively. The first four can either perform the opposite action (*a* if a was performed or a if *a* was performed) or the new process (X or Y) could evolve, which leaves the other process running on its own. From the synchronized processes, either synchronization happens (leading to 0) or it evolves to leave only one process, which then evolves to 0.

      The flow diagram is simpler: either use the usual way of showing P with four ports, x, y, a and *a*, or show the internal processes (a.X, *a*.X, a.Y, *a*.Y) with the appropriate ports.

   e. Hiding a means that all but the τ transitions are lost. The flow diagram shows the process P with only the x and y ports visible.

f. Q = a.Z|*a*.Z

Q evolves to Z|Z either directly via synchronization or by both processes evolving separately. As Z = X + Y, this is the opposite of the last question with the τ transition coming before the choice rather than after (i.e. Q = a.(X + Y)|*a*.(X + Y)

4.

a. A = a.P | b.Q

A can simulate B, but B can't simulate A as something else could occur when either the P or Q process occurs in parallel with b.Q or a.P respectively.

Chapter 11. Lecture 7 - 24th of October, 2003, 10am

11.1. An Infinite Process

P = a.P | *a*.P. From P, three things can happen: the a can synchronize with another process, the *a* can synchronize with another process, or both can synchronize together. On syncrhonization, we end up with P|P, effectively doubling the number of processes. The same problem occurs on any action, as any of the three possibilites reveal a P process (P|*a*.P and a.P|P are the other alternatives), which is equivalent to what we had at the start. This leads to an infinite answer. A bisimiliar, but not equivalent, answer is to use a single P process and have the three actions (a, *a* and τ) looping on this process.

11.2. Alternatives

What follows are some alternatives to CCS and the Pi Calculus. There will be no formal questions on these alternatives.

- Communicating Sequential Processes (1980ish)
- Petri Nets (1969)
- Mazurkowics Traces (1974)

11.2.1. Petri Nets

Petri nets were the first model of concurrent systems. Councurrency means sharing resources. Petri nets look at each system individually. Transitions are marked on diagrams by a | symbol. Each token, passing between processes, has to trigger the transition simulataneously. Coloured Petri nets allow differentation between processes. A | transition can then require five green tokens and a red token, for example.

With the neighbour/burgular scenario, if the neighbour interaction has taken place, then there is no token for the neighbour to interact with. CCS only allows two processes to interact at any one time, but Petri nets allow any number of processes to trigger a transition. However, Petri nets are hard to reason about. It also isn't possible for the neighbour to feed the cat, while the house is being burguled.

Petri nets can model the infinite process P shown above. The Petri net representation is infinite in tokens, rather than being infinite pictorally. A single token generates two tokens, two tokens generate four. So each transition doubles the number of tokens, just as CCS doubles the number of each process.

Chapter 12. Lecture 8 - 24th of October, 2003, 11pm

12.1. Alternatives

12.1.1. Trace Theory

The two paths, $a.0|b.0 \longrightarrow^{a} b.0 \longrightarrow^{b} 0$, and $a.0|b.0 \longrightarrow^{b} a.0 \longrightarrow^{a} 0$ are not necessarily different. If things happen simulateneously, the order you write them down in is irrelevant. (3.4) + 5 can be written 345, 354 or 534.

If A represents the alphabet, and *i* represents independency, $\forall ab.a\ i\ b \longrightarrow b\ i\ a$. With traces, things can be switched if they are independent. With the trace, abcdecba, if daec are independent (i.e. a *i* c, a *i* d, a *i* e, d *i* c, etc.), then strings such as abcdceba are equivalent.

But, how do you tell if traces are equivalent if part of one is lost? For example, if we have the trace abc, how do we tell that abdce is equivalent if d and e are lost? For these traces to be equivalent, the first must contain a and d.

12.1.2. CSP

CSP was created around the same time as CCS and is similiar, but with no inputs and outputs. Actions that occur simulatenously have a separate picture for each. CSP provides a way of writing down Petri nets

12.1.3. Process Metrics

For the process P = a.P, d^{∞} is the behaviour. The metric, d, requires that $dx{:}x \longrightarrow \Re$, $d(xy) \geq 0$ and $d(x,y) = 0$ if $x = y$. For $ab \longrightarrow a^2b \longrightarrow a^3b \longrightarrow$ etc., we need to know if they are independent. If they are, ba is also a possibility. The behaviour is a^{∞} or $a^{\infty}b$ (the b is irrelevant, as a goes on forever). Why is a^5b closer to $a^{\infty}b$ than a^4b? The initial stuff is not important; only the limiting behaviour is important.

In the Fibonacci sequence (1,1,2,3,5,8,13,21,...), each number divided by the one before gives $1 \div 1$, $2 \div 1$, $3 \div 2$, $5 \div 3$, $8 \div 5$, $13 \div 8$, $21 \div 13$, etc. This pattern converges on $(1 + \sqrt{5}) \div 2$ which is ~ 1.7. To prove this wrong, it has to be proved that there is a range outside this area. a^5b and a^4b disagree on the 4th letter, while a^3b and a^4b disagree on the 3rd letter. Finding where they disagree gives the distance. $2^{-(\text{where they disagree})}$ gives a distance that gets smaller and they converge on 0.

12.2. CCS

$P \cap P$ links processes P and P. The two processes connect along a channel, which joins the output of one P to the input of the other. The remaining input and output of the pair form the input and output of the composite process.

A scheduler could join multiple P processes together in a more complex example.

Chapter 13. Tutorial 5 - 28th of October, 2003, 3pm

13.1. Sheet 4

1.

   a. Yes, as $0 + 0 \approx 0|0$

   b. No. If P = a.0 and Q = b.0, P + Q would result in either a single a or b action going to the process 0, whereas P|Q would complete two actions (a then b, or b then a) before reaching the zero process.

2.

   a. With weak bisimulation, it doesn't matter about things you can't see. For example, $\longrightarrow^{a} \longrightarrow^{\tau}$ is the same as $\longrightarrow^{a}$, and $\longrightarrow^{a} \longrightarrow^{\tau} \longrightarrow^{b}$ is equal to $\longrightarrow^{a} \longrightarrow^{b}$. In strong simulation, if P can do something and Q simulates it, then Q can also do the same thing, leading to a situation at least as powerful. Weak simulation doesn't check for τ transitions, and only the visible operation is considered.

   b. τ.(a + b) and (τ.a) + (τ.b) are not strongly bisimiliar, but are weakly bisimiliar. The choice can't be seen. Similiarly, a can be weakly simulated by $\tau^{m}a\tau^{n}$

Chapter 14. Lecture 9 - 31st of October, 2003, 10am

14.1. Semantics

Semantics describe what things mean. How do we know 5 + 2 = 7? Every symbol has to be explained. This has to be built up from the axiom 0 + 1 + 1 = 2 (i.e. writing 0 + 1 + 1 is the same as writing 2). Bisimiliarity shows that two processes have the same meaning. P = a.0|b.0 is defined as an a followed by a b, or a b followed by an a. A set of diagrams exists containing all processes with the same meaning.

5 + 2 = 7 tells us that 5 * (5+2) can be replaced by 5 * (7). This is replacing equals with equals, and is particular to numbers. It doesn't generally happen with algebra. We know a.(P+Q) $\neq$ a.P + a.Q, so the same rule as for numeric addition doesn't apply. τ.(a.0 + b.0) and τ.a.0 + τ.b.0 are weakly bisimiliar, but, in practice, we can always choose either button in one but not the other. It is only equivalent observationally, and not during interaction. The same applies for a + b and a + τ.b.

Congruences occur when equals can be replaced with equals. If A $\equiv$ B, then anything done with A has to lead to equivalent things in B. This leads to A + X $\equiv$ B + X and A|X $\equiv$ B|X.

14.2. The π Calculus

Imagine the scenario where there are three processes, P, Q and R. P is connected to Q by the channel, b, and P is connected to R by the channel, a. P can send messages to both Q and R, but what happens if P wants Q to send a message to R? The communication channel, a, moves between processes. The π calculus allows both pictoral representation and algebraic manipulation of this.

P = *b*P.P'.

Q = b(x).

The above equations show the process P being passed along the communication channel, b. However, passing the process allows access to other communication channels other than a. In the π calculus, only links are passed over and not processes. Initially, P = *a*5.P' and R = a(x).R'. For Q to act as an intermediatary in this communication, it needs to know the value and the link to be able to send the message. Redefining P as P = *b*a.*b*5.P' and Q as Q = b(name).b(val).*name*val.Q' allows the passing of the value and the link to Q. Q doesn't know what it will receive, but knows that the must output the received value on the received link. R is still a(x).R', and is unaware of the change.

Initially, the only possible synchronization is between P and Q to transmit the link. P and Q then synchronize again to transmit the value. Finally, Q and R synchronize to transmit the value. The π calculus allows named channels as well as data to be sent. The rights to use the channel are transmitted to Q to allow it to send the message.

14.2.1. A Real Life Scenario

The boss of a company is meant to ring the bank, but is busy and delegates the task to his secretary. The boss has a communication channel with the secretary (the intercom) as well as a telephone number for the bank (the 'callBank' channel). In the above example, P would be the boss, Q would be the secretary and R would be the bank. Responsibility can be seen to be passed across from the boss to the secretary. The boss uses the intercom to tell the secretary he needs to call the bank, and about the request he needs to send.

Boss = *intercom*callBank.*intercom*compReq.Boss

The secretary doesn't know what comes from the boss, only that it will be a telephone number and some information.

Secretary = intercom(number).intercom(info).Secretary'

The secretary then has to use the number to send the information.

Secretary' = *number*info.Secretary"

Whether or not the boss or the secretary calls the bank, the bank waits for a call on its line to receive the request.

Bank = callBank(request).Bank'

Assuming synchronization only takes place, the following occurs:

Table 14-1. Synchronizations

Boss	**Secretary**	**Bank**	**Variable Values**
intercom.callBank	intercom(number)		number = callBank
intercom.compReq	intercom(info)		info = compReq
	*callBank*compReq	callBank(request)	request = compReq

As we know, we can prevent operations other than synchronization by hiding actions.

Chapter 15. Lecture 10 - 31st of October, 2003, 11am

15.1. Notation

Hiding notation has changed over the years. The following all hide the a channel in the process, P.

- P\{a}
- P\a (1980's)
- (a)P (1989 - π calculus)
- (new a)P (2001ish)

15.2. Scope/Boundness/Restriction/Free Variables

F(x) = $\int$f(x,y) dy = $\int$f(x,w) dw = $\int$f(x,z) dz, etc. y is a *dummy* variable and can be replaced by anything *but* x (which leads to $\int$f(x,x) dx). $\int$(x,y) = ^{2} = $xy^3 \div 3$, while $\int$f(x,x) dx = $\int x^3$ dx = $x^4 \div 4$.

Similarily, λx.λy.(x-y) = λx.λw.(x-w) = $\lambda x \lambda$z(x-z), etc. ((((λx.λy.(x-y))3)-1) causes every x to be replaced by 3, which equals ((λy.(3-y))-1) which leads to 3 - (-1) = 4. y again can't be replaced with x, because this gives ((((λx.λx.(x-x))3)-1) = ((λ3.(3-3))-1) = (-1) - (-1) = 0.

The choice of letters is the problem. The scope of x is λy.(x-y). You can't replace the x with a variable when within its scope. x is said to be bound. With P = a(x).*b*(x).P', the x tells us that the same value applies in both places. For a(x).P, P is the *scope* of x. With P\a, P is in the *scope* of a.

Take S = τ(a.0 + b.0). A c could exist along with τ, giving S = (*c*.(a.0 + b.0)|c.0)\c, with τ being the synchronization of the c's. Replacing b with d still gives the same picture. Substitution is written as S{d/b}. c can't be substituted for b, as it is hidden. It is already in scope and bound.

Chapter 16. Tutorial 6 - 11th of November, 2003, 3pm

16.1. The Mobile Phone Network

From a quick brainstorm, the following things could be considered relevant in a model of a mobile phone network:

- Towers, Phones, Main Tower
- Links between towers and phones
- Signals moving
- Passing of communication channels
- Capacity of towers
- May not to be able to transfer between towers
- Battery expires
- Mobile phone checks which aerial it is near
- Towers have an ID signal
- "Places"
- Overlap between tower signals
- How do towers communicate with each other?

An example scenario, where the phone decides to switch between towers, could then be as follows (with the possibility of a "busy" response):

1. Monitor my signal strength
2. When low, broadcasts to nearby towers to find out who’s near
3. We work out the next tower to use and confirm availability
4. Both towers talk to the user, only one is heard. The main tower receives both signals and decides which to relay.
5. Tower 2 is told I’ve been using Tower 1.
6. Tower 2 tells Tower 1 it’s ready to take over.
7. Tower 1 cuts off the connection.

Chapter 17. Lecture 11 - 14th of November, 2003, 10am

17.1. CCS and the π Calculus

What can CCS do?

CCS

- $P \longrightarrow^{\alpha} P'$
- P + Q
- P|Q
- 0
- a.0
- *a*.0

What about the π calculus?

π **Calculus**

- P :: = 0
- *x*y.P
- x(y).P
- τ.P
- new x P
- [x=y] P (not required)
- P|P
- P + P (not required)
- $A(y_1...y_n$ (not in 2002)

*x*y.P $\longrightarrow^{xy}$ P

x(y).p $\longrightarrow^{x(val)}$ P{val/y} (y is a dummy variable)

If P = y(u).0, then x(y).P = x(y).y(u).0. If the value passed in y is 2, then x(y).y(u).0 $\longrightarrow^{x(2)}$ 2(u).0 $\longrightarrow^{2(value)}$ 0.

τ is a silent internal action. τ.P $\longrightarrow^{\tau}$ P. new x P $\equiv$ P \ {x} can do everything that P can do that doesn't involve an x i.e. P $\longrightarrow^{\alpha}$ P' implies that new x P $\longrightarrow^{\alpha}$ new x P' if $\alpha \neq$ x or *x*.

Take the equation, new x(in(x).x(y).*y*(out).0). The x and *y* operations can't take place, because they are guarded. So, it follows that new x(in(x).x(y).*y*(out).0 $\longrightarrow^{in(val)}$ val(y).*y*(out).0 $\longrightarrow^{val(val')}$ *val'*(out).

With new x(x(y).P), we can't do anything as the x is hidden. With new x(x(y).P|*x*z.Q), the x's can synchronize, allowing z to be substituted for y in P i.e. new x(x(y).P|*x*z.Q) $\longrightarrow^{\tau}$ new x(P{z/y}|Q).

17.2. Adam and Bill

Adam's task is as follows:

1. Think of a number
2. Tell it to Bill, then go shopping.

Bill's task is:

1. Take Bill's number
2. Do anything to it (multiply it by 2, for example)

Adam = *r*5.Adam'

Bill = r(num).Bill'(num)

Running the two in parallel, hiding r (effectively, a local variable):

new r(Adam|Bill)

As r is hidden, only synchronization is left. This leads to new r(Adam|Bill) $\longrightarrow^{\tau}$ new r(Adam'|Bill'(5){5/num}).

17.3. Rules So Far

1. *x*y.P $\longrightarrow^{xy}$P
2. x(y).P $\longrightarrow^{x(y)}$ P
3. τP $\longrightarrow^{\tau}$ P
4. If P $\longrightarrow^{xw}$ P', Q $\longrightarrow^{x(y)}$ Q', this implies that P|Q $\longrightarrow^{\tau}$ P'|Q'{w/y}
5. If P $\longrightarrow^{\alpha}$ P' | Q $\longrightarrow^{\alpha}$ Q', this implies that P|Q $\longrightarrow^{\alpha}$ P'|Q', and the same in reverse.

The exam focuses on usage, rather than formulae.

Chapter 18. Lecture 12 - 14th of November, 2003, 11am

18.1. Adam retires

Adam's tired! So, he gets Caroline to do it instead. Adam tells Caroline the number and the link (robert or r to his friends), via mike.

Adam = *mike*s.*mike*r.Adam'

Caroline = mike(num).mike(chan).*chan*(num).Caroline'

a and *a* doesn't really mean anything. One is input, and the other is output, but which is which is just a matter of convention. The π calculus doesn't make a distinction when passing links. It can be assumed that the relevant end is passed to the receiver. Therefore, things like $\longrightarrow^{x(y)}$ never happen. x(y).*y*z.0 is perfectly acceptable. The links are names; their behaviour depends on the equation.

But, how do we make Caroline use the number with robert, and not another link, such as stephen? Adam|Bill|Caroline $\longrightarrow^{\tau}$ *mike*r.Adam' | Bill | mike(chan).*chan*5.Caroline' $\longrightarrow^{\tau}$ Adam' | r(num).Bill'(num) | *r*5.Caroline' $\longrightarrow^{\tau}$ Adam'|Bill(5)|Caroline'

π calculus is about getting other people to do work for you. Caroline gains knowledge of robert over the course of the transaction. The channel and data can be passed in either order. So, how do we deal with the problem of link/data mismatch? Hiding robert has no effect on Caroline - the private link is passed to her, but she doesn't know that it's private.

18.1.1. Molecular Actions

A new private communication channel is created for the transmission.

Adam = new chan(*mike*chan.*chan*r.*chan*5.Adam')

Caroline = mike(w).w(x).w(num).*x*num.Caroline' $\longrightarrow$ chan(x).chan(num).*x*num.Caroline'

A shortcut for creating the new channel is Adam = *mike*(chan).*chan*<r,5>. Order matters for matching the input.

18.2. More π Calculus Constructs

With the process, [x=y]P, it acts like P if x = y, otherwise it acts like 0. This allows the equivalent of a switch

statement. For example, [a(val).([val=x]P + [val=y]Q + [val=z]R)]. If y is received and replaces val, {y/val} = [[y=x]P + [y=y]Q + [y=z]R) = 0 + Q + 0 = Q. This is useful, but is not generally used. A more complicated way is available for purity. There is no test for inequality! In π calculus, P|Q and P+Q are the same as in CCS.

Chapter 19. Tutorial 7 - 18th of November, 2003, 3pm

19.1. Sheet 5

1.

   a. a.0|b.0 is a concurrent process. The transition graph shows the two possible routes of $\longrightarrow^{a}$ b.0 $\longrightarrow^{b}$ 0 and $\longrightarrow^{b}$ a.0 ^{a} 0.

   b. (a.0 + b.0) + (a.0|b.0) is a concurrent process, as it involves processes running in parallel. There are four possible transitions from the process: a, a, b and b. Both one of the a's and one of the b's leads to the zero process. The others (from the parallel process) lead to the opposite process (a $\longrightarrow$ b.0 and b $\longrightarrow$ a.0). Another transition would lead either of these to the zero process as well. The diagram is effectively the combination of a.0 + b.0 and a.0|b.0.

2. f can simulate e, but bisimulation fails as 0 and b.0 can not both simulate each other. Simulation is a relationship between processes. If something can be done in X, it can also be done in Y. If Y simulates X, this produces pairs (P_f,P_e) such as (0,b.0), where one process goes to 0 and the other to b.0 on the same transition. How do we prove something not bisimiliar? For a simulation, S = {(P_f,P_e), ...}, the simulation must contain the pair (0, b.0), where 0 simulates b.0. Reversing the arrows gives S^{-1} = {(P_{e,P^f}, ...} with (0, b.0) as a member (the pairs are swapped). But, 0 doesn't simulate b.0, so the bisimulation fails.

3.

   a. P = a.a.(*b*.0 + *c*.0). The transition diagram for this process is a simple run of two a transitions, and then either a b or c transition leading to a zero process.

   b. These are just simple subsitutions. P_1 = u.u.(*v*.0 + *w*.0).

   c. P_2 = v.v.(*w*.0 + *u*.0).

   d. P_3 = w.w.(*u*.0 + *v*.0

   e. CCS simply uses a different hiding operator, so Sys = (P_1 | P_2 | P_3) / {v,w}

   f. For a flow diagram of Sys, P_1 is connected to P_2 on the u and v channel, and P_3 on the w channel. P_2 is connected to P_1 on u and v, and P_3 on w. It thus follows that P_3 connects to both P_1 and P_2 on w (it having the only input w), P_1 on u and P_2 on v.

   g. The process in full is u.u.(*v* + *w*)|v.v.(*w* + *u*)|w.w.(*u* + *v*). The zero processes are omitted for clarity. The only possible transition from the start is for P_1 to use up it's u transitions. u is the only channel not hidden, and the only one that can synchronize with anything outside Sys. Following this, there are two possibilities depending on the application of the + in P_1. Either P_1 synchronizes with P_2 to give v.(*w* + *u*) | w.w.(*u* + *v*) or it synchronizes with P_3 to give v.v.(*w* + *u*) | w.(*u* + *v*)

Chapter 20. Lecture 13 - 21st of November, 2003, 10am

20.1. The Mobile Phone Network

To model the mobile phone network, we start by guessing and then find out what's missing. The behaviour of an agent depends on its channels. The phones connect to the tower currently in use by the signal and talk channels, and to all accessible towers via the broadcast channel. They also have negotiation channels for negotiating a new connection to a tower. The towers themselves have two channels to the main tower, one a control channel and the other a talk relay.

We can define the phone as:

Phone(signal,talk,negotiation,broadcast) = talk(words).Phone + *talk*(words).Phone + signal(talk_relay, control).BroadcastingPhone(signal,talk,negotiation,broadcast,talk_relay,control).

This means that the phone can either send or recieve words, or begin the process of changing tower, by becoming a BroadcastingPhone on receipt of talk_relay and control channels.

The signal needs to tell the phone the channels to drop. We need to define the BroadcastingPhone to handle tower changes:

BroadcastingPhone(...) = *broadcast* negotiation.BroadcastingPhone(...) + talk(words).BroadcastingPhone(...) + *talk*(words).BroadcastingPhone(...) + negotiation(new_talk, new_signal).SwitchingPhone(..., new_talk, new_signal).

The BroadcastingPhone continues transmitting words, while it sends out a negotiation link along the broadcast channel. When it receives a reply on its negotiation channel, it becomes a SwitchingPhone. A tower which is currently idle (IdleTower) communicates with the BroadcastingPhone and is defined as follows:

IdleTower(...) = *broadcast*(negotiation).*negotiation* (talk, signal).NegotiatingTower(..., negotiation).

Following communication between the BroadcastingPhone and the IdleTower, the BroadcastingPhone receives information on the IdleTower's talk and signal channels, and both change, becoming SwitchingPhone and NegotiatingTower respectively. A SwitchingPhone sends back the information on its talk_relay and control channels to the tower it has received a reply from, while continuing to transmit words (sending along both talk links this time).

SwitchingPhone(signal,talk,negotiation,broadcast,talk_relay,control,new_talk, new_signal) = *negotiation*<talk_relay, control>.WaitingPhone + talk.SwitchingPhone(...) + *talk*.*new_talk*.SwitchingPhone

Meanwhile, the NegotiatingTower waits for further information from the phone. On receipt, it sends the

information to the MainTower via control. It then waits for confirmation from the MainTower, at which point it tells the Phone that it has taken over, and becomes a normal Tower.

NegotiatingTower(...) = negotiation(new_talk_relay, new_control).*control*<new_talk_relay, new_talk_control>.control.*negotiation*.Tower

The WaitingPhone simply waits for the tower to confirm that it has now taken over transmission, and then resumes being a normal Phone.

WaitingPhone(signal, talk, negotiation, broadcast, talk_relay, control, new_talk, new_signal) = negotiation.Phone(new_signal, new_talk, negotiation, broadcast)

Finally, we just need to finish off by defining a normal Tower (one which forwards words from a phone, until told not to via control) and the MainTower.

Tower(talk, signal, talk_relay, control) = talk.*trelay*.Tower(...) + *talk_relay*.talk.Tower(...) + *signal*<talk_relay, control>.Tower(...) + control.IdleTower

The Main Tower either relays talk_relay communications between towers, or allows new towers to take over via the control channel. The example below assumes three towers, 1 through 3, with 1 and 3 communicating with each other and 2 taking over from 1.

MainTower($talk\_relay_1$, $control_1$, $talk\_relay_2$, $control_2$, $talk\_relay_3$, $control_3$) = $talk\_relay_1$.*$talk\_relay_3$*.MainTower(...) + $talk\_relay_3$.*$talk\_relay_1$*.MainTower(...) + $control_2$($talk\_relay_1$, $control_1$).*$control_1$*.MainTower($talk\_relay_2$, $control_2$, $talk\_relay_3$, $control_3$, $talk\_relay_1$, $control_1$)

In the final option, the order of the parameters of MainTower are changed, so that Tower 2 swaps position with Tower 1.

Chapter 21. Lecture 14 - 21st of November, 2003, 11am

21.1. Structural Equivalence

P|Q ≡ Q|P is a structural congruence, as is P + Q ≡ Q + P and P | 0 ≡ P. Brackets are irrelevant, as P|(Q|R) ≡ (P|Q)|R and P + (Q+R) ≡ (P+Q) + R. Also, new x (P|Q) ≡ P|new x Q if x ∉ fn(P). For new x P, x is "bound" in P. a(x).P also has x bound in P. If a name isn't bound, it may be free. The free names in P are fn(P). A name is free if it is used in a process, and is not either an input variable or hidden. In *b*x.0, all names are free. fn(*b*x.0) = {b,x} and fn(a(x).*x*b) = {a,b}. x is not in the free names of the latter, as it is bound as an input variable.

21.2. Reverse Diagrams

How do we work back from 0|0, given two transitions from the initial process and then a further transition from each intermediate process to 0|0? First, we give the nearest arrow labels (there is no synchronization, so the labels can't sync and one is not the output of the other). From here, the rest is obvious, in this case, and we have the process a.0|b.0.

21.3. Replication vs. Recursion

A = a.A, P = a.Q, and Q = b.P. Everything is defined recursively in CCS. But, recursion can get messy e.g. A = a.A + b.c.A. Replication says to keep creating copies. This is easily defined recursively as:

REPEAT-P = P | REPEAT-P

!P ≡ REPEAT P. !P ≡ P|!P. If P $\longrightarrow^{\alpha}$ P', then !P $\longrightarrow^{\alpha}$ P'|!P.

Replication is redundant. It can be replaced by recursion. Choosing between two processes can be done using triggers:

new x(*x*.0 | x.P | x.Q)

The processes can only synchronize, as x is hidden. x is ∉ fn(P,Q) so, if we choose P,:

⟶ new x(0|P|x.Q) ≡ new x(P|x.Q) ≡ P|new x(x.Q) ≡ P.

Similiarly, choosing Q:

⟶ new x(0|x.P|Q)

| can also be made redundant, just as NAND is the only logical gate needed. Replication can also replace recursion, using triggers. If P = A|b.0, where A $=_{def}$ x.A, then we can easily see that if A is made equal to !x.0, this is equivalent to x.0|!x.0 ≡ 0 | !x.0 ≡ !x.0. Thus, P ≡ !x.0|b.0.

A more general method is also possible. Taking P again, we can see that in P_1 = ... P_1 ... P_1 ... P_1 ..., P_1 itself occurs at least once (this is the nature of recursion).

If we pick a name that is not yet used, e.g. "p", then, given any expression involving P_1, we replace P_1(*x*x) with *p*(*x*). Finally, if you want to model X = ... P_1..., replace it by $X^{\wedge\wedge}$ = new p($x^{\wedge}$ | !P(*x* | $RHS^{\wedge}$.

In our example, X = P and P_1 = A. A = x.A and P = A|b.0. RHS = x.A. We choose the letter 'a' for A, which gives $P^{\wedge\wedge}$ = new a($P^{\wedge}$ | !a.$RHS^{\wedge}$) ≡ new a(*a* | b.0 | !a.x.*a*) ≡ new a(*a* | b | a.x.*a* | !a.x.*a*) ⟶ new a(b | x.*a* | !a.x.*a*) $\longrightarrow^{x}$ new a (b | *a* | !a.x.*a*)

We end up back where we started, so it is the same process.

21.4. Reaction

P $\longrightarrow^{\tau}$ P' ≡ P"' can be written as P ⟶ P". For example, a.P|*a*.Q $\longrightarrow^{\tau}$ P|Q ≡ Q|P can be written as a.P|*a*.Q ⟶ Q|P. Algebraically, these are different statements.

Chapter 22. Tutorial 8 - 25th of November, 2003, 3pm

22.1. Sheet 6

1.

   a.

   b.

   c.

   d. The reactions are simply the synchronizations.

      new y (x y | y x)) | x(u).u w

      $\longrightarrow$ new y(y x|y w)

      The big difference between CCS and π calculus is that the hidden y can be passed to processes outside its initial scope.

2.

   a. new x(x | x.P | x.Q) "$\equiv$" τ.P + τ.Q $\longrightarrow$ new x(P | x.Q) $\equiv$ P|new x(x.Q) $\equiv$ P or $\longrightarrow$ new x(x.P | Q) $\equiv$ new x(x.P) | Q $\equiv$ Q.

   b. The above is P + Q, when the internal actions are ignored.

   c. Take $a_1P_1 + a_2P_2 + a_3P_3 ... a_mP_m \mid b_1Q_1 + b_2Q_2 + b_3Q_3 ... b_nQ_n$. If a_2P_2 and b_nQ_n are chosen, they can both react independently or they may be able to synchronize. That is, = $a_2.(P_2 \mid b_nQ_n) + b_n (a_2P_2 \mid Q_n) + [a_2 = b_n] \tau.(P_2|Q_n)$. Thus, $\Sigma a_i.P_i \mid \Sigma b_jP_j = \Sigma\{a_i.(P_i \mid b_j.Q_j) + b_j.(a_iP_i \mid Q_j) + [a_i = b_j] \tau.P_i \mid Q_j\}$. The | can be removed, but a horrible recursive definition results.

3.

   a. S = a.b.P + b.a.P, where P = a.b.P. Suppose we do a, then b. a.b.P $\equiv$ a.b.a.b.P. P can do a and b, but we're now stuck at P = a.b.P. P is a trigger to start expanding on.

      a.X is guarded (it can't become X without soaking up a). a.X|a allows this. To trigger something, we guard it with the input and trigger it by running it in parallel with the output.

      S becomes a.b.p + b.a.p, if we change P to be the trigger, p. Running this in parallel with !p.a.b.p allows replication to replace the recursion.

For 3a, $P = E_P$, $A = E_A$ and $B = E_B$. x is used as the trigger for a and y is used as the trigger for b. Thus, $Gen_A = !x.E_A$ and $Gen_B = !y.E_B$. This will produce infinitely many copies of A or B, when it receives an *x* or a *y*.

$P^A = (E_P \mid Gen_A \mid Gen_B = A \mid B \mid !x.a.A \mid !y.b.B$. We replacec A with *x* and B with *y* to give *x* | *y* | !x.a.*x* | y.b.*y*, and hide the new x and y channels. This results in new x,y (*x* | *y* | !x.a.*x* | !y.a.*y*.

b. Q = P|a(u).*u* b|a(v).*v* c. There is nothing recursive in Q, so it is left as is. P = ! new x (*a* x.*x* y). Replication is trivially produced by recursion, by running the process in parallel with itself i.e. P = new x (*a* x.*x* y)|P.

Chapter 23. Lecture 15 - 28th of November, 2003, 10am

23.1. Data Types and Structures

23.1.1. Boolean Values

There are two possible Boolean values: true and false. In the π calculus, Boolean operations ask the value if it has the right sort of value to operate on. Effectively, the operator asks "I want to operate on you. I need to know whether you're true or false.". More formally, this is:

Bool Op = *i*<tf>.(...)

i is an interface channel for interfacing with the value. Following on from this, we can define processes for True and False:

True = i(t,f).*t*

False = i(t,f).*f*

Both processes receive both a true and a false channel from the operator. Depending on which value they are, they reply on the appropriate channel. Using this information, we can make our operator do something with this information, effectively emulating a simple 'IF T THEN P ELSE Q' statement.

IF(P,Q) = *i*<t,f>.(t.P + f.Q)

To prevent interference, we need to hide the channels from the outside world.

(new t,f) (*i*<t,f>(t.P + f.Q)

Running our if statement in parallel with True gives us an appropriate reaction:

i(tf).*t* | (new t,f) *i*<t,f>(t.P + f.Q)

$\longrightarrow$ (new t,f) (*t*|(t.P + f.Q))

$\longrightarrow$ (new t,f) P

This works, provided we choose t,f to be new (i.e. $\notin$ fn(P)). We can also define a simple not function. A not function inverses the value of a Boolean. If B = true, then not B = false. Likewise, if B = false, then not B = true.

NOT IF(P,Q) can be defined simply by swapping the triggers of P and Q:

NOT IF(P,Q) = (new t,f) *i*<t,f>(f.P + t.Q)

To define NOT itself, we need to provide a process that receives a Boolean value and returns the opposite:

NOT = (new t,f) *i*<t,f>(t.False(i) + f.True(i))

An operation is run in parallel with a value to see what comes out. Running NOT with True evaluates as follows:

NOT|True = (new t,f) *i*<t,f>(t.False(i) + f.True(i)) | i(t,f).*t*

$\longrightarrow$ (new t,f)[*i*<t,f>(t.False(i)+f.True(i)) | i(x,y).*x*] (x and y are substituted for t and f in True and False, so that we can extend the scope of new)

$\longrightarrow$ (new t,f)[(t.False(i) + f.True(i))|*t*]

$\longrightarrow$ (new t,f)[False(i)] $\equiv$ False(i) as t,f $\notin$ fn(False).

23.1.2. Enumerated Data Types

Other enumerated data types, such as the days of the week, can be created and used in the same way as Booleans. The data value recieves the channels (the possible values) from the function, and responds on its value.

Mon(i) = i(Mon,Tues,Wed,Thu,Fri,Sat,Sun).*Mon*

Tue(i) = i(Mon,Tues,Wed,Thu,Fri,Sat,Sun).*Tue*

Wed(i) = i(Mon,Tues,Wed,Thu,Fri,Sat,Sun).*Wed*

Thu(i) = i(Mon,Tues,Wed,Thu,Fri,Sat,Sun).*Thu*

Fri(i) = i(Mon,Tues,Wed,Thu,Fri,Sat,Sun).*Fri*

Sat(i) = i(Mon,Tues,Wed,Thu,Fri,Sat,Sun).*Sat*

Sun(i) = i(Mon,Tues,Wed,Thu,Fri,Sat,Sun).*Sun*

By using triggers, we can also define a CASES syntax for our new data types. This works in the same way as the IF function above. The function queries the data value and recieves a response which acts as a trigger for the appropriate process.

CASES(P_{Mon}, P_{Tue}, P_{Wed}, ...)(i) = *i*(mon,tue,wed,thu,fri,sat,sun).(mon.P_{Mon} + tue.P_{Tue} + wed.P_{Wed} + ...)

23.1.3. Lists

Lists can be built up using a Root process and several Node processes, each of which has a channel to a Value process. A Root process on its own specifies an empty list. Functions interact via the interface channel, and can't see the internal structure. Storing True in a list would be represented as:

(new v)(True(v) | (new p)(Node(ivp)|Root(p)))

The p channel connects the node to the root of the list. The v channel connects the node to the value (True) and the i channel connects to the list to the outside world.

Chapter 24. Lecture 16 - 28th of November, 2003, 11am

24.1. Data Types and Structures

24.1.1. Lists

Representing a Node as a flow diagram shows it having three channels: i (the interface), v (the value) and p (the parent). A Root node only ever has one channel: this is either the p channel connecting it to the first Node in the list, or the interface channel if the list is empty. Independent processes only join together via channels when they happen to meet.

If the list is not empty, functions communicate with a Node. Otherwise, communication occurs with the Root. What happens depends on which process the interaction takes place with.

Root(i) = i(rn).*r*

Node(ivp) = i(rn).*n*<vp>

This test takes place in the same way as the test on Booleans and enumerations. The value is probed via the interface channel, and is given access to two channels: the root channel and the node channel. The value responds on the appropriate channel. In the case of the node, it also sends back links to its value process and its parent process.

To add items to the list, we need a Cons function. This can be defined as follows:

Cons(V,L)(i) ≡ (new p) (L(p) | Node (ivp) | V(v))

This joins the three processes (the existing list, L, the new node, N, and the new value, V) together. Flow graphs are linked to the structure of the data.

24.1.2. Natural Numbers

We can represent integers in a similiar, but simpler, way to lists. We use a Zero node to represent Zero, and Succ to represent a successor. Zero is equal to the value 0, and Zero|Succ is equal to the value 1. It then follows that 2 is represented as Zero|Succ|Succ. Running things in parallel is the same as executing a function.

Zero(i) = i(zs).*z*

Succ(ip) = i(zs).*sp*

Again, the data values respond on the appropriate channel. Succ includes a link to its parent in the response. We can represent One and Two as follows:

One(i) = (new p)(Zero(p)|Succ(ip))

Two(i) = (new p)(One(p)|Succ(ip))

Using this information, we can easily construct an AddOne process:

AddOne(N)(i) = (new p)(N(p) | Succ(ip))

To add two numbers, we need to join the two sets of nodes together.

24.2. A Little More π Calculus

P(X) = ... is a definition. P<chan> = <chan, chan, ack> is instantiation. Inputs are always bound variables.

24.3. Topics Covered

By lecture:

1. FSMs, Turing machines, X-Machines, Hypercomputing, Jobshop
2. Flow and transition diagrams, bisimulation
3. Bisimulation, Buffer scenarios
4. More bisimulation, Petri Nets, Trace Theory
5. Understanding the tutorial paper
6. Reading Week
7. Formal structure of the π calculus, Mobile Phone network
8. Agent syntax, structural equivalence, matching, recursion and replication, mobile phone network implementation.
9. Data structures
10. Revision
11. Revision
12. Reading Week

What is there to know?

- Simulation and Bi-simulation
- Flow and transition diagrams

- Physical modelling
- Data stuff

24.4. Data Ordering

P = *x*<a,b>.P' ≡ *x*<a>.*x*<b>.P'

Q = x(uv).Q' ≡ x(u).x(v).Q'

R = x(u,v).R' ≡ x(u).x(v).R'

The problem is that outputs are split up. A private channel is negotiated first.

P = (new w) *x*<w>.*w*<a>.*w*<b>P

Q = x(w).w(u).w(v).Q'

R = x(w).w(u).w(v).R'

x<a,b> is the abbrievation for this process.

24.5. Unary Abstraction

One is a function:

One = (i)((new i)(...))

This is how functional calculus is turned into π calculus.

Chapter 25. Tutorial 9 - 2nd of December, 2003, 3pm

25.1. Sheet 7

1.

a. P = (a + b)

b.

- P = a.0, Q = b.0
- P = b.0, Q = a.0
- P = (a.0|b.0), Q = 0
- P = 0, Q = (a.0|b.0)

c. Since neither P nor Q uses τ explicitly, the τ-action in the diagram must come from an (x-*x*) synchronization; actually it could come from ANY restricted synchronization, but we may as well use "x" because that's already restricted.

Since we're starting with a composition of processes, we should think of the final "0" as being "0|0". That way, we have the same number of components at each stage.

Working backwards from 0|0, we need to get to this stage by either a or b. Therefore, one of the processes running in parallel must be a.0+b.0. Both a and b must be in a single process, as the two processes are running in parallel, and have to evolve singularly. For one to be a.0 and the other to be b.0, there would have to be an intermediate process after either an initial a or b. As the diagram shows, we don't do both a and b, but one or the other, so the + operator is used.

From a.0+b.0|0, we need to get back to our starting process of (new x)P|Q via a synchronization. As our processes are running in parallel, and the only possible transition is τ, the synchronization must be via a hidden channel. As we already have x restricted, we can use this on both sides of the | to produce a synchronization. This gives us P = x.(a.0+b.0) and Q = *x*.0, or vice versa, as the solution.

d. This question is the same as the last, except for the fact that two synchronizations are needed. As both τ actions are the only transitions, both involve hidden channels. We can solve this by simply extending our previous answer to involve another synchronization on x, following our choice of a.0 + b.0. This gives P = x.(a.x.0+b.x.0) and Q = *x*.*x*.0 as one possible solution. Other solutions exist, as there is no reason for both *x* actions have to come from the same process, or for P to contain the a and b actions.

2.

a. Both processes act as possible values by communicating their type along the appropriate channel given by the query on the interface channel, i. Zero responds via z to indicate its value, while a node (a

non-zero value) responds via the n channel, with a link to its parent (Zero for 1, another Node for all other numbers).

b. In both cases, the Node has two channels: the interface channel and the parent channel. It accepts queries from functions via the interface channel. The parent channel is an internal channel which links the Node to its parent. In the case of One, this is Zero, while with Two, the parent is One. Thus, Two is actually made up of three processes, two of which are Nodes and one of which is Zero.

Two(i) = (new p)(Node<ip>|One(p))

≡ (new q)(Node iq)|One(q))

≡ (new q)(Node (iq)|(new p)(Node (qp)|Zero(p)))

≡ (new pq)(Node (iq)|Node (qp)|Zero(p))

c. If we look at how the processes representing 1 and 2 are built up, we can see that 1 is constructed through running a Node in parallel with 0, and that 2 is constructed by running a Node in parallel with one. Succ(N)(i) is a generalisation of this, where a Node is run in parallel with an existing number, N. As can be seen from the existing examples of 1 and 2, this results in the existing number + 1. Therefore, succ(n) models n + 1 successfully.

d. To find the result of a monus operation on two numbers, we first need to know how to, given a number, we find the preceding number (the equivalent of n - 1, where n is the number we are given). In other words, we need to find out how to subtract one from a number, before we find how to subtract one number from another.

Pred(N)(i) will result in N(i) if N(i) is Zero(i) (i.e. the predecessor of 0 is 0, as we don't represent negative numbers), and M(i) if N(i) = Succ(M)(i) (i.e. we get the preceding node, if N is a successor of another node).

Thus, for our predecessor function, we need to know whether we are dealing with a zero node or a node that has a predecessor. We already have a way of doing this through the interface channel and our cases function:

Cases(P,Q(x))(i) = (new nz)(*i*<nz>.(z.P + n(x).Q(x)))

Applying a function in process calculus involves running two linked processes in parallel. For our predecessor function, we need Cases(P,Q(x))(i)|Zero(i) to return zero.

Cases(P,Q(x))(i) | Zero(i) ≡ (new zn)(*i*<nz>.(z.P + n(x).Q(x))) | i(xy).*x* ⟶ (new zn)[(z.P + n(x).Q(x) | *z*)] ⟶ (new zn)[P|0] ⟶ P.

Therefore, we want P to be Zero. What happens if we run Cases(P,Q(x))(i) in parallel with N(ip), a node?

Cases(P,Q(x))(i) | N(ip) ≡ Cases(P,Q(x))(i)|(new p) (Node(ip) | M(p)) ≡ (new znp) [*i*<zn>.(z.P + n(x).Q(x) | Node(ip) | M(p))]

⟶ (new znp) [(z.P + n(x).Q(x) | *n*<p> | M(p) ⟶ (new znp) [Q(p) | M(p)].

So, expanding Cases running with a Node gives us our Q(p) running in parallel with the predecessor of N(p), M(p). However, the p channel is hidden and we need an interface channel to access the predecessor, as M only expects to receive information on i. In effect, what we want is M(i). To achieve this, we need to make Q act as a transparent entity between functions using the predecessor and M, providing an interface channel from which it takes input to feed on to M. This gives us:

Q(c)(i) = i(ab).*c*<ab>.0

Q simply takes input on i, and passes it on to M, via the channel, c. Therefore, we now have our Pred(N)(i) function:

Pred(N)(i) ≡ N(i) | Cases(P, Q(c))(i) where Cases(P, Q(c))(i) ≡ (new zn)*i*<zn>.(z.Zero(i) + n(x).i(ab).*x*<ab>)

The main thing that makes the predecessor function difficult is trying to get access to a hidden node. The monus function simply uses Pred recursively to find the predecessor of each of the two nodes, until one node reaches zero.

Chapter 26. Lecture 17 - 5th of December, 2003, 10am

26.1. 2000 - 2001 Exam Paper

1.

2.

a. The superscripts and subscripts are irrelevant. The superscript merely specifies that the buffer can handle two inputs, and the subscripts just save on definitions.

$B^{(2)}(x_0, x_1, y_0, y_1) =^{def} x_0.B_0^{(2)} \langle x_0, x_1, y_0, y_1\rangle + x_1.B_1^{(2)} \langle x_0, x_1, y_0, y_1\rangle$

B_i represents both B_0 and B_1, and is defined as follows:

$B_i^{(2)}(x_0, x_1, y_0, y_1) =^{def} y_i.B^{(2)} \langle x_0, x_1, y_0, y_1\rangle + x_0.B_{0i} \langle x_0, x_1, y_0, y_1\rangle + x_1.B_{1i} \langle x_0, x_1, y_0, y_1\rangle$

B_{ij} represents four processes: B_{00}, B_{10}, B_{01} and B_{11}. It is defined as follows:

$B_{ij}(x_0, x_1, y_0, y_1) =^{def} y_j.B_i^{(2)} \langle x_0, x_1, y_0, y_1\rangle$

From this, we can determine that $B^{(2)}$ has two output channels, y_0 and y_1, and two input channels, x_0 and x_1. From $B^{(2)}$, input can be recieved on either of the two x channels. This lead to appropriate B_i processes, where i depends on which x channel input was received. From here, the input can be output on the corresponding y channel, or further input can be recieved via the x channels. If the input is output, we return to the starting point of $B^{(2)}$. Otherwise, we move to B_{ij}, with j being equal to the previous value of i. From here, our only option is to output one of the two stored values, taking us back to the intermediate B_i stage.

$C(x_0(x_1, y_0, y_1) =^{def} x_0.C_0 \langle x_0, x_1, y_0, y_1\rangle + x_1.C_1 \langle x_0, x_1, y_0, y_1\rangle$

$C_0(x_1, y_0, y_1) =^{def} y_0.C \langle x_0, x_1, y_0, y_1\rangle$

$C_1(x_1, y_0, y_1) =^{def} y_1.C \langle x_0, x_1, y_0, y_1\rangle$

$P \cap Q =^{def} \text{new } m_0, m_1 (\{m_0, m_1 / y_0, y_1\} P \mid \{m_0, m_1 / x_0, x_1\} Q$

new $m^{\longrightarrow}$ means that all m's are new. $\{m_0, m_1 / y_0, y_1\}$ means to substitute y_0 for m_0 and y_1 for m_1.

$C \cap C = C(x_0, x_1, m_0, m_1) \mid C(m_0, m_1, y_0, y_1)$

C ∩ C is a pair of C processors, with substitutions made to allow the outputs of one to link to the inputs of another. A normal C process takes an input on one of its two x channels, and outputs it on the appropriate y channel. C ∩ C substitutes the y channels of the first C process point for a hidden channel, which is also substituted for the x channels in the second C process.

A flow graph of C ∩ C thus shows the two processes linked by the m channels, which replace the y and x channels of the two processes. A transition graph of C ∩ C shows the possible transitions that can take place. Initially, input must be recieved from one of the x channels. Once this has occurred, synchronization takes place which passes the input from the first C process to the second. Only synchronization is possible, as the m channels are hidden. From here, two things can happen. As the second process is now unguarded (the synchronization uses the action that was x in the original process), it can become the appropriate C_i process, where i is the channel the input was passed in on. From here, it can output the value via its appropriate y channel. This leaves both processes back in their original states.

Alternatively, the first C process can receive further input. At this point, the second process must output the first input to allow operations to continue. When it does so, we end up back at the choice outlined above, as the second input is synchronized across the m channels.

b. The second C process becomes U, which only has one output channel. Otherwise, the flow graph is exactly the same as before.

c. We don't cover inference trees! The transition is basically the synchronisation which leads to the input already in C (making it C_0) being moved across the m channel, m_0, to U, which becomes U_0.

d. C ∩ U, without synchronization, become a variant of $B^{(2)}$, without the second y channel (y^1). The second y channel is the equivalent of the broken channel in U. Thus, the transition graph of C ∩ U running as a sequential process is the same as that for B^2, except that the y_1 actions disappear, and the path from the initial action, x_1 leads to a dead-end (as values input on x_1 can never be output). As y_1 no longer exists, this also means that BU_{01} and BU_{11} equate to the zero process, as the only possible action for these processes is guarded by y_1.

$BU^{(2)}(x_0, x_1, y_0) =^{def} x_0.BU_0^{(2)} \langle x_0, x_1, y_0\rangle + x_1.BU_1^{(2)} \langle x_0, x_1, y_0\rangle$

$BU_0^{(2)}(x_0, x_1, y_0) =^{def} y_0.BU^{(2)} \langle x_0, x_1, y_0\rangle + x_0.BU_{00} \langle x_0, x_1, y_0\rangle + x_1.BU_{10} \langle x_0, x_1, y_0\rangle$

$BU_1^{(2)}(x_0, x_1, y_0) =^{def} x_0.0 + x_1.0$

$BU_{i0}(x_0, x_1, y_0) =^{def} y_0.BU_i^{(2)} \langle x_0, x_1, y_0\rangle$

26.2. Scope Expansion and Substitution

((new znd)[*i*<zud>.u(x).*x*]) | i<xyz>.*y*(p)

We can’t extend the restriction, new znd, as z <8714> fn i<xyz>.*y*(p). Therefore, we need to substitute a different channel name:

$\equiv$ ((new znd) [*i*<zud>.u(x).*x*]) | i<xyz>.*y*(p)

$\equiv$ (new znd)[(*i*<zud>.u(x).*x* | i<xyz>.*y*(p))]

$\longrightarrow$ (new znd)[u(x).*x* | *u*(p)] $\longrightarrow$ (new znd)[*p*]] $\equiv$ *p*.

Chapter 27. Lecture 18 - 5th of December, 2003, 11am

27.1. Modelling The Internet

The Internet involves sending and receiving information between computers. But, what are the agents involved? The main agents are the computers involved and the packets they transmit. During the simple task of retrieving a web page, a client computer transmits a request to a server, via a finite number of relays. The server replies to the client with the web page, which is then displayed. Our Computer process is thus one of the three agents, Client, Relay and Server.

Computer = Client + Relay + Server

We know need to model the request to the web server. The client transmits a request via its output channel (which we'll call *a*). Input from a is received by a relay, which passes it on to either another relay or, eventually, the server. The client becomes a WaitingClient and waits for information to be returned from the server.

Client(a) = *a*<info>.WaitingClient(a)

The relay's sole job is to forward information. Information can arrive on either channel x or y, and is output on the opposite channel to which it arrived.

Relay(x,y) = x(info).*y*<info>.Relay(x,y) + y(info).*x*<info>.Relay(x,y)

The waiting client recieves information, displays it and then becomes a Client again, capable of making requests. For the purpose of this simulation, we assume that no further requests are made until the response from the first is received. By doing this, our model more accurately simulates an individual applicatin on the client rather than the client as a whole. In real-life, the possibility for multiple requests would depend on the application issuing the request. Multiple requests can be handled by allowing another Client process as an option, but we then need a way of tracking requests.

WaitingClient(a) = a(info).*display*.Client(a)

The display could be more acurately modelled by a Monitor agent:

Monitor = display.Monitor

The server is the most complex process of the three. At its simplest, the server receives the request, processes it, outputs the response and becomes again.

Server(d) = d(infoaddress).*d*<info>.Server(d)

There is no need to model the lookup. It could be a large matching database such as:

= [infoaddress="index.html"]*d*<info.index.html> + [...]*d* + [...]*d*

But, this server process will get stuck. Following one request, it will not be able to make any more until the first is dealt with. A more reliable version is:

Server(d) = d(infoaddress).BusyServer(d, infoaddress)

BusyServer(d, infoaddress) = Server(d) + *d*<info>

But, here we have the same problem as we would with allowing multiple client requests; the server loses track of what is being requested. Alternatively, we can model our client and server as a Sender and Receiver process running in parallel.

Client(sender, receiver) = Receiver(rec) | Sender(send)

Receiver(rec) = rec(x).Reciever(rec)

Sender(send) = *send*<addr>.Sender(send)

A better version of the sender would use a hidden channel for returning the response.

Sender(send) = (new addr)(*send*<addr>.Sender(send)

A real-life server would recieve requests and forward them onto a new process, which deals with the request and returns the response, while the main server continues recieving requests.

Server(d) = (new internal) d(infoaddress).*internal*<d, infoaddress>.Server(d)

InternalProcess(d, internal) = internal(infoaddress).process.*d*<info>.0

A new InternalProcess is used for each request.

27.2. Unary Abstraction

P(x) = x(a);... ≡ P = (x)(x(a)...). The latter is a unary abstraction of the former. The typing in past exam papers e.g. z: CHAN(E) can be ignored.

Chapter 28. Tutorial 10 - 9th of December, 2003, 3pm

28.1. 2000 - 2001 Exam Paper

1.

a.

i. $Q =^{def} a.(b.c.0 + b.d.0).$

$R =^{def} a.b.(c.0 + d.0).$

We need to show that R simulates Q (Q S R). Symbolically, the simulation, S, is made up of the following pairs:

- (Q,R)
- (b.c + b.d, b.(c + d))
- (c, c + d)
- (d, c + d)
- (0, 0)

Both Q and R follow a single a action at the start. The difference between the two is that, when Q follows this with a b action, it is left with either c or d. However, when R does the same, it still has both c and d to choose from.

ii. Now we need to show that $Q \not\equiv R$ by pairings. If S is a simulation of Q by R, with the pairings shown above, then S^{-1} can not be a simulation. We already have a simulation S, which specifies that for every action, α, that Q does, R can match it. For Q and R to be equivalent, we need to be able to reverse the pairs in S, which gives S^{-1}. To prove that they are not equivalent, we therefore just need to reverse each pair and stop when they are obviously not reversible. This occurs in this simulation with either of the pairs (c, c + d) and (d, c + d). These are the pairs that make Q and R different. Reversing the pairs gives (c + d, c) and (c + d, d), which suggests that either of c or d on their own can simulate c + d. This is obviously not the case, so S^{-1} is not a simulation.

b.

i. Given $P = (\text{new } x) ((x.Q_1 + y.Q_2) \mid \tau.x.0) \mid (x.R_1 + y.R_2)$, we need to show that this can be rewritten as $P = (\text{new } z) (((y.\{z/x\}Q_2 + z.\{z/x\}Q_1) \mid (y.R_2 + x.R_1)) \mid \tau.z.0)$. We can do this simply by using the rewrite rules.

Firstly, we need to extend the scope of the hiding operator, new, to cover the whole equation. However, here there is a problem. x is being hidden in the first part of the equation, but another x is used in the later part. Therefore, we need to substitute x for something else in either of these parts.

Looking at our goal, it is clear that z appears as a substitution for x in the first part of the equation (turning new x into new z). This gives us our first part of the answer.

$\equiv$ (new z) ((z.{z/x}Q_1 + y.{z/x}Q_2) | τ.z.0) | (x.R_1 + y.R_2)

We can now extend the scope of the hiding to cover the later part of the equation:

$\equiv$ (new z) ((z.{z/x}Q_1 + y.{z/x}Q_2) | τ.z.0 | (x.R_1 + y.R_2))

What remains is just a simple matter of re-arranging the individual parts of the equation, by using rules such as P + Q $\equiv$ Q + P and P | Q $\equiv$ Q | P, so as to match our goal. Firstly, our goal statement has the first of the three processes running in parallel starting with y.{z/x}Q_2. In our current version, z.Q_1 kicks off the first process. Simple use of the P + Q $\equiv$ Q + P rule allows us to swap the two.

$\equiv$ (new z) (y.{z/x}Q_2 + z.{z/x}Q_1) | τ.z.0 | (x.R_1 + y.R_2))

However, the statement still differs from our goal. Two final changes are needed. First, another use of the + rule, to rearrange the third part of the equation:

$\equiv$ (new z) ((y.{z/x}Q_2 + z.{z/x}Q_1) | τ.z.0 | (y.R_2 + x.R_1))

And, finally the P | Q $\equiv$ Q | P rule is used to swap the second and third process around.

$\equiv$ (new z) ((y.{z/x}Q_2 + z.{z/x}Q_1) | (y.R_2 + x.R_1) | τ.z.0)

The term 'normal form' suggests that everything should be covered by the scope of a new statement to the left, via enough rearranging.

ii. We don't cover inference diagrams.

iii. There are four possible reactions from P'. Two are obvious, and involve the synchronization of the y and z pairs. The others are simple x and y actions, which are possible because these channels aren't hidden, unlike z.

Chapter 29. Lecture 19 - 12th of December, 2003, 10am

29.1. Data Structures

We have already looked at three possible data structures in detail: the Boolean values (with True and False processes), the natural numbers (constructed by running Zero and n Node processes in parallel, where n is the number to represent) and lists (constructed from a Root node running in parallel with one Node for each element, each of which has an associated Val process). These data structures only ever have one channel visible to the outside world. To operate on them, we need to know which type of process we are talking to. Thus, the definition of a data structure is always the same. We communicate via the interface channel and each node tells us what it is and what it is connected to.

Queries always take the form $X_n = i\langle x_1, x_2,...,x_n\rangle$ with the response being an output from the value on the appropriate channel, $x_n\langle...\rangle$. With the numbers, Zero(i) = i(zn).z and Node(ip) = i(zn).$n\langle p\rangle$. So, X_{17} = i(x_1, x_2, ..., x_n).$x_{17}\langle ab\rangle$

A function is executed by running it in parallel with the data structure.

$F(X) = F \mid X_n \longrightarrow Y$

X_n expects a query request of the form i(x_1,...,x_n), so:

$F = i\langle x_1, x_2x_2, ..., x_n\rangle.(x_1 (..).F_1 + x_2 (..).F_2 + ... + x_n (..).F_n$

Each F_n defines what occurs, given a certain type of node.

29.1.1. Integers

With integers, either 0 = Zero(i) or n = Node(ip). There are only two options in this model, so the query is $i\langle zn\rangle$.

$F(N)(i) = i\langle zn\rangle.(z.F_{zero} + n(p).F_{node})$

29.2. 2000 - 2001 Exam Paper

1. See Tutorial 10.
2. See Lecture 17.
3.

4. Ignore anything that doesn't mean anything, such as sorting. Zero $=^{def}$ (k).k(z,s).*z* is the same as Zero(k) $=^{def}$ k(zs).*z*. It can be renamed as Zero(i) $=^{def}$ i(zs).*z* or even Zero(i) $=^{def}$ i(zn).*z* for familiarity.
   a. Simple book work definitions of π calculus operations.
   b. Succ(N)(i)n takes the same form as F(N)(i) defined above. Therefore, Succ(N)(i) takes the form:

      $Succ(N)(i) = i\langle zn\rangle.(z.F_{zero} + n(p).F_{node})$

      All we have to do is work out what F_{zero} and F_{node} are. If the sucessor is operating on zero (z.?), then it should become One(i). For Nodes, N becomes N + 1. In both cases, Succ(N)(i) turns into a Node, as One(i) = Node(ip)|Zero(p).

      First, we'll see what happens if we run Succ(N)(i) in parallel with Zero(i) (i.e. execute the function Succ on Zero), with the two processes linked by the interface channel.

      $Succ(i)|Zero(i) \equiv i\langle zn\rangle.(z.F_{zero} + n(a).F_{node}) \mid i(zn).z$

      $\longrightarrow (z.F_{zero} + n(p).F_{node}) \mid z$

      $\longrightarrow F_{zero}$

      We want Succ(N)(i) to return One(i) when run with Zero(i), so F_{zero} must be One(i) = (new p) Node(ip)|Zero(p).

      Now, we need to look at Succ(N)(i) working with a Node(ip). In this case, we'll work with N + 1 to ensure that we are not operating on Zero.

      $Succ(N+1)(i) = Succ(i) \mid \text{"N + 1"}(i) \equiv i\langle zn\rangle.(z.F_{zero} + n(a).F_{node}) \mid (new\ p)(Node(ip)|N(ip))$

      Extending the scope, we get:

      $\equiv (new\ p)\ (i\langle zn\rangle.(z.F_{zero} + n(a).F_{node}) \mid Node(ip)|N(ip))$

      Expanding Node(ip) (with appropriate substitution) gives us:

      $\equiv (new\ p)\ (i\langle zn\rangle.(z.F_{zero} + n(a).F_{node}) \mid i(rs).r\langle p\rangle|N(ip))$

      $\longrightarrow (new\ p)\ ((z.F_{zero} + n(a).F_{node}) \mid n\langle p\rangle|N(ip))$

      $\longrightarrow (new\ p)[F_{node} \mid N(p)]$

      From Succ(N)(i)|Node(ip), we want two nodes connected to N. What we currently have is a connection to N, via the parent channel, p, running in parallel with F_{node}. Therefore, our F_{node} should be the two

nodes which connect to N.

F_{node} = [Node(iq) | Node(pq)]

Chapter 30. Lecture 20 - 12th of December, 2003, 11am

30.1. 2000 - 2001 Exam Paper

1. See Tutorial 10.
2. See Lecture 17.
3.
4.
    a. See above.
    b. Succ(N)(i) = i<zn>.(z.(new p)[Node (ip)|Zero(p)] + n(p).(new q)[Node(ip)|Node(qp)])

    Succ uses up the original Node which is running in parallel with N. The data is destroyed, and has to be replaced. This is why two nodes are returned in F_{node}. By talking to something, we destroy it. The solution is to use replication to generate copies of the process to interact with.

    c. Sorting is not covered.
    d. Cases(P,F)(k) = (new zs)*k*<zs>.(z.P + s(p).F(p))

    P is a process used when Cases is run with Zero(i), and F is a process used when Cases is run with Node(ip), with p pointing to the predecessor. This question simply requires us to show how the Cases function evolves, when run in parallel with Zero(i) and Node(ip).

        i. Cases(P,F)(k) | Zero (k) ≡ (new zs)(*k*<zs>.(z.P + s(p).F(p))) | k(zs.*z*

        ≡ (new zs)[*k*<zs>.(z.P + s(p).F(p)) | k(zs.*z*]

        ⟶ (new zs)[z.P + s(p).F(p) | *z*]

        ⟶ (new zs)[P] ≡ P as required

        ii. This may not work, as there are infinitely many choices of Succ(N)(k) that could be defined in part (b).

        Cases(P,F)(k) | Succ(N) (k) ≡ (new zs)(*k*<zs>.(z.P + s(p).F(p))) | k<zn>.(z.(new p)[Node (kp)|Zero(p)] + n(p).(new q)[Node(kp)|Node(qp)])

        ≡ (new zs)(*k*<zs>.(z.P + s(p).F(p))) | (new pq) (k<zn>.(z.[Node (kp)|Zero(p)] + n(r).[Node(kr)|Node(qr)]))

        ≡ (new pqzs)(*k*<zs>.(z.P + s(p).F(p)) | k<an>.(a.[Node (kp)|Zero(p)] + n(r).[Node(kr)|Node(qr)]))

We know that Succ(N)(k) is simply a Node running in parallel with N, so we can simplify the equation. To get the same, we would need to follow all the successor reactions.

$\longrightarrow^{*}$ (new pqzs) (*k*<zs>.(z.P + s(p).F(p)) | (new p') [Node (kp') | N(p')]

≡ (new pp'qzs) (*k*<zs>.(z.P + s(p).F(p)) | [Node (kp') | N(p')]

≡ (new pp'qzs) (*k*<zs>.(z.P + s(p).F(p)) | [k(xy).*y*<p'> | N(p')]

⟶ (new pp'qzs) [(z.P + s(p).F(p)) | *s*<p'> | N(p')]

⟶ (new pp'qzs) [F(p')| N(p')]

≡ (new p') [F(p') | N(p')] as required.

30.2. Final Notes

30.2.1. Associative and Commutative Rules

P | Q ≡ Q | P and P + Q ≡ Q + P are commutative, as is m + n ≡ n + m.

If an operator is associative, the brackets are irrelevant i.e. (P | Q) | R ≡ P | (Q | R) and (P + Q) + R ≡ P + (Q + R).

30.2.2. Monadic

x(p) and *x*<a> are input and output respectively.

30.2.3. Polyadic

x(pqr) and *x*<abcdefg> have more than input and output value.

30.2.4. Unary Abstraction

P is a process which depends on one channel (unary).

30.2.5. Final Words

There will be no pure maths stuff this year, as in last year’s paper.

Appendix A. GNU Free Documentation License: Version 1.2, November 2002

A.1. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.3. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not

use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- 0. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

A.7. 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.9. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

A.10. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no

Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.