Prof. Dr. Ir. Joost-Pieter Katoen
apl. Prof. Dr. Thomas Noll
B. Kaminski, H. Wu, S. Chakraborty

Concurrency Theory WS 2013/2014

— Concurrency Workbench Bonus Tutorial —

Hand in until January 7 before the exercise class.
All points are bonus points.

---

1 Introduction

By now you know how to model a system as a CCS (Calculus of Communicating Systems) process, check whether two processes are equivalent (bisimilar, weakly bisimilar) and have an elementary grasp of $L\mu$ calculus (basically Hennesy–Milner Logic with recursion). In this tutorial we will use the Concurrency Workbench to have some hands on experience with concurrent systems (hopefully enjoy it in the process). For this tutorial we have chosen the Edinburgh Concurrency Workbench.

2 Installation

Edinburgh cwb is easiest to start with. Follow the instructions

1. Go to the website : http://homepages.inf.ed.ac.uk/perdita/cwb/getting.html
2. Navigate through the hyper link "First, download the source code, heap image or executable you want" at the top of the page.
3. Enter your email address.
4. Download the executable: "Stand–alone executable for Linux on x86 (Redhat 5.?) (T)CCS"
5. Change the file to an executable if need be ($ chmod +x ⟨filename⟩).
6. $ ./⟨filename⟩
7. Voila!

If everything goes according to the plan you should be greeted by "command:". Make sure that the process algebra is CCS. We are now in business. You can now define systems in CCS, define propositions in modal logic, perform model checking and most notably interactively simulate the behaviour of a CCS process.

3 Getting started

Once you get it running, the most important thing you must first learn is how to quit! Type "quit;" and it safely exits with a polite remark. At this point you should go through the user manual (but that is no–joy). Let's digress from that for the moment, and learn it through examples. To start with, we should be able to define processes and actions. The processes will be called *agents*, and they will be named with the first letter in capitals (e.g B). Names of actions, on the other hand, will start with lowercase letters (e.g in), and output actions start with \ (e.g $\overline{out}$).

Let us start with a simple semaphore Sem = get.$\overline{\text{put}}$.Sem. Equivalent code in CCS is:

```
Command: agent Sem = get.'put.Sem;
Command: print;

** Agents **
agent Sem = get.'put.Sem;
```

What do you reckon this process will do? You can simulate this process to see that it behaves according to your expectation.

```
Command: sim Sem;

Simulated agent: Sem
Transitions:
    1: ---- get ----> 'put.Sem

[0]Sim: 1;

  ---- get ---->

Simulated agent: 'put.Sem
Transitions:
    1: ---- 'put ----> Sem
[1]Sim: 1;

  ---- 'put ---->

Simulated agent: Sem
Transitions:
    1: ---- get ----> 'put.Sem

[2]Sim:
```

Type "quit;" when you had enough. Four more constructs is all we need to brave most of the problems. Non–deterministic choice "+", parallel composition " | ", renaming "[x/y]" and restriction "$\backslash L$". Lets quickly get an example of each.

```
Command: agent B = Sem[in/get,out/put];
Command: agent A = B|Sem;
Command: agent D = A\{out,put};
Command: agent C = A + D;
```

Run "sim" on each of them to see the state space. Type "clear;" when you want to forget everything so far and start afresh. You are now warmed up for the following.

Exercise 1 (3-Place Buffer and Weak bisimulation) (1* Point)

Write the CCS sentences for a sequential three–place Buffer B_s and a parallel three–place Buffer B_p (refer to the lecture slides) and check whether they are *weakly bisimilar* ($B_s \approx B_p$), using the Command $eq(B_s, B_p)$; □

4 The Logic

Next thing on the agenda is to write specifications in $L\mu$. Though the semantics of $L\mu$ is rather convoluted, we will begin with simpler, more intuitive formulae first. Let's start with the simplest of all formulae, "True" and "False".

```
Command: prop P = T;
Command: prop Q = F;
```

The modal operators $[\alpha]$ and $\langle\alpha\rangle$ work the usual way. Recall the three–place sequential buffer from Exercise 1. Let us see that it is indeed a three–place buffer.

```
Command: prop P = <in><in><in>T;
Command: prop Q = <in><in><in><in>T;
```

To check whether a process A satisfies a formula P, use the command "checkprop(A, P);". See what happens when you try to check the above property on the three–place buffers. It not only gives you an answer true or false, but also in its hubris, challenges you to a *game*. Similarly, for weak $[\alpha]$ (or $\langle\alpha\rangle$), use the operator $[[\alpha]]$ (or $<< \alpha >>$ respectively). You now know the following:

- An agent A satisfies $[K]P$ if for all $A \xrightarrow{a} A'$, where $a \in K$, A' satisfies P (could be vacuously true).
- An agent A satisfies $< K > P$ if there exists a $A \xrightarrow{a} A'$, where $a \in K$, A' satisfies P.
- An agent A satisfies $[[K]]P$ if for all $A \stackrel{a}{\Longrightarrow} A'$, where $a \in K$, A' satisfies P.
- An agent A satisfies $<< K >> P$ if there exists a $A \stackrel{a}{\Longrightarrow} A'$, where $a \in K$, A' satisfies P.
- Boolean operators are negation, conjunction, disjunction and implications are $\sim P$, $P\&Q$, $P|Q$ and $P => Q$, respectively.

Exercise 2 (Simple HML) (1* Point)

Define the following property for your three–place buffer: A buffer can only output something if it has input something. Check this property for every state of the sequential buffer. Write a weak version of the property and check it on parallel buffer. □

The perceptive of you by now have noticed the limitations of formulae restricting to above constructs only. That brings us to the esoteric subject of recursion. Let us learn it through examples. Suppose we want to express the fact that all the time some property P holds (invariant). Another way of saying this property is that if this property holds then P also holds and every successor also shares this property. That is,

$$X \implies P \wedge [Act]X,$$

a little fix-point theory jargon would give us the $L\mu$ formula: $\nu X.(P \wedge [Act]X)$ (or as in the lecture: $X \stackrel{\max}{=} P \wedge [Act]X$). How about a property P finally holds in some path? It could be thought as a property that holds if P holds now or it will hold is some successor. Precisely,

$$P \vee \langle Act\rangle Y \implies Y,$$

the $L\mu$ formula being $\mu Y.(P \vee \langle Act\rangle Y)$ (or as in the lecture: $Y \stackrel{\min}{=} P \vee \langle Act\rangle Y$) (a dual of the previous). It is given by the following command

```
Command: prop Q = max(X.P & [-]X);
Command: prop R = min(Y.P | <->Y);
```

Exercise 3 (Gedankenexperiment) (1* Point)

Think of $L\mu$ formulae for the following properties: (1) There exists a path where property P always holds. (2) For all paths a property P finally holds.[1] (3) Does there exist a Deadlock state reachable from the start state. (4) Safety.[2] □

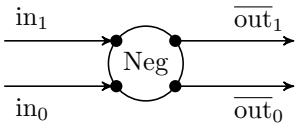
5 Examples

CCS is a good language to specify the labeled transition system for your process and the specification. How hard do you imagine model checking is? The next exercise will give you some idea.

[1] There is an easy way and a right way to do this, the right way is to argue about your answer from the first principles.
[2] Spoiler alerts: don't refer to the slides at first!

Exercise 4 (Boolean) (2* Points)

We intend to define boolean algebra in CCS. Think of input variables as communication channels one for bit 1 or 0. Let me help out a bit. The process for negation is as follows,



$$\text{Neg} = \text{in}_1.\overline{\text{out}}_0.\text{Nil} + \text{in}_0.\overline{\text{out}}_1.\text{Nil}$$

Now the onus is on you to define processes for boolean operators And and *Or*. Thus, any Boolean formula can now be defined as a process. Define a process for $(a \wedge \neg b)$ using the processes for *And* and *Neg*.

Now write a process explicitly defining $a \wedge \neg b$ (just like the negation). For behavioural equivalence you should opt for *observable trace equivalence*. This is given by command : mayeq(P1,P2); You should also try out dftrace(P1,P2); This produces a sequence of observable actions that can be taken by one state but not the other. □

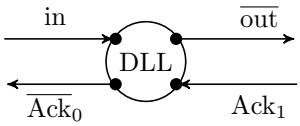
In case you have not thought of this yourself, here is a gentle nudge.

Exercise 5 (NP-Hard Extended) (2* Points)

You would have noticed the problems of checking satisfiability for formulae having repetition of variables. Tweak your processes to handle this. In essence, write a process for $a \wedge \neg a$ and check that, in all paths it never gives output 1 (refer Exercise **??**). □

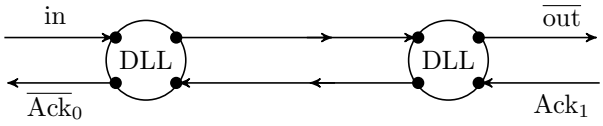
Let us talk about protocols on the Data Link Layer (DLL) for communications. Here, we will consider the most abstract portrait of DLL. Packets (messages or acknowledgements) are sent to DLL from (Network) layers from above which are then transmitted across the communication channel to the receiver and any packet received is send to the layers above. We'll only examine protocols for sending and receiving.

We can model the receiver and sending layer as follows:



$$\text{DLL} = \text{in}.\overline{\text{out}}.\text{DLL} + \text{Ack}_1.\overline{\text{Ack}}_0.\text{DLL}$$

Consider full-duplex connection, we define a protocol Persistent-acknowledgements Retransmissions (PAR) where a message is sent and an acknowledgement is received. The CCS model that we have in mind is:



PAR = ?

Exercise 6 (PAR) (2* Points)

Please do the following:

1. Write a CCS process (PAR) for it .
2. Give a $L\mu$ formula showing that PAR can repeatedly transmit any number of messages and acknowledgements.
3. Explain with a $L\mu$ formula, how PAR fails to do this sometimes. In other words show that it is not reliable.
4. Instead of full-duplex, argue that in half-duplex channels the problem disappears. In other words, amend the definition of DLL to rectify the anomaly.

□

The next protocol we will study is the *alternating bit prolocol*. The sender sends a message with a bit 0 (or 1). When the receiver gets the message with a 0 (or 1) bit, it sends an acknowledgement with a 0 (or 1 respectively)bit. When the sender receives a 0 (or 1) bit acknowledgement, she changes to message with 1 bit (or 0, respectively). We give you a full duplex lossless channel C.

```
agent C = min0.Cm0 + ain0.Ca0 + min1.Cm1 + ain1.Ca1;
agent Cm0= 'mout0.C + ain0.Cm0a0 + ain1.Cm0a1;
agent Ca0= 'aout0.C + min0.Cm0a0 + min1.Cm1a0;
agent Cm0a0 = 'mout0.Ca0 + 'aout0.Cm0;
agent Cm0a1 = 'mout0.Ca1 + 'aout1.Cm0;
agent Cm1a0 = 'mout1.Ca0 + 'aout0.Cm1;
agent Cm1a1 = 'mout1.Ca1 + 'aout1.Cm1;
agent Cm1= 'mout1.C + ain0.Cm1a0 + ain1.Cm1a1;
agent Ca1= 'aout1.C + min0.Cm0a1 + min1.Cm1a1;
```

Here channel min0 is reserved for input message with bit 0 , aout0 is reserved for output-ing acknowledgement with bit 0, so–on–so–forth. For example state Cm0a1 means channel C has both message with 0 bit and ack with 1 bit in it.

Exercise 7 (Alternating Bit Protocol) (2* Points)

Design the sender and the receiver over this channel. Check for Deadlocks

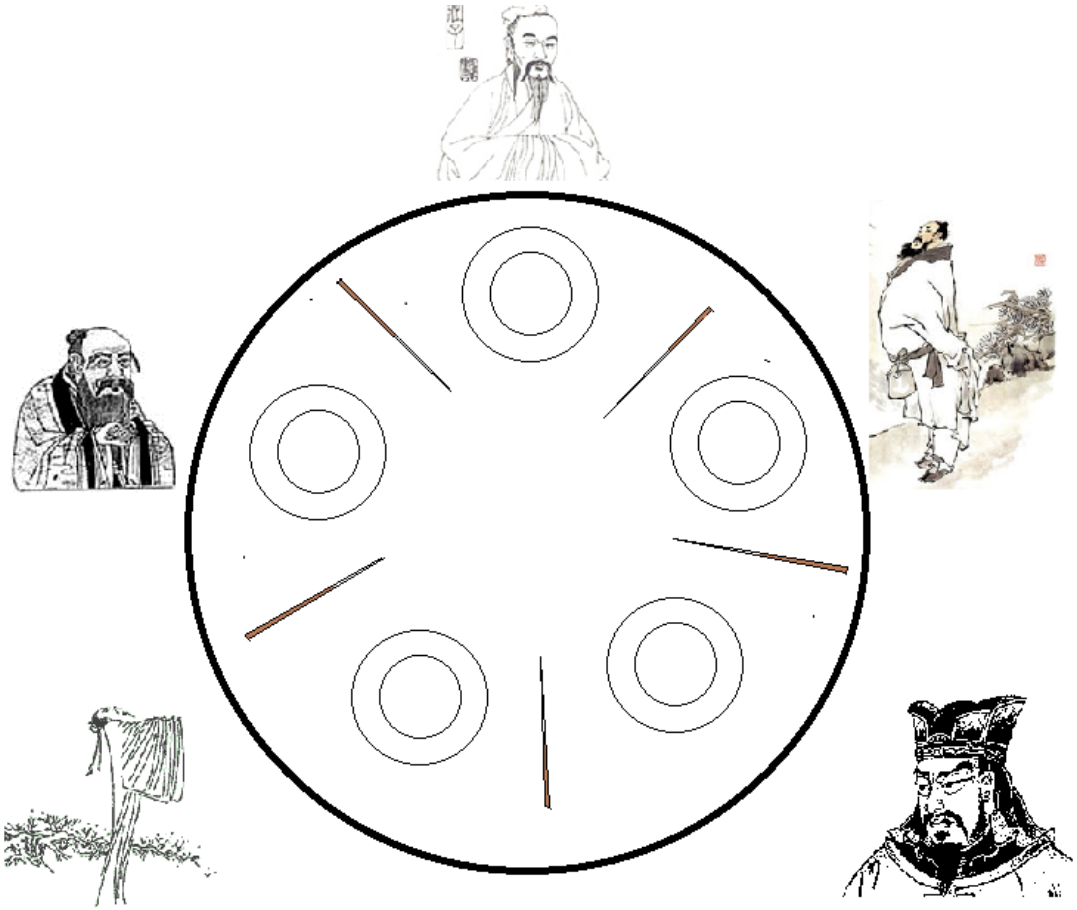
Assume "smsg" is an external input (packet) to the sender from network layers and the receiver outputs "rmsg" to Network layers above DLL. Check, for every execution if "smsg" is send then "rmsg" is received.

Now consider a lossy channels, i.e, packets may be dropped. Redesign (if need be) so that the protocol never get stuck. Check for deadlocks and verify, whether there is an execution (instead of all) where send messages are received. □

Last but by no means the least, we will consider the Dinning philosopher problem. It brilliantly captures the design dilemmas for concurrent algorithms. We surmise most of you have heard this problem before. We write the problem description for the unfortunate ones. Here it goes.

There are five philosophers Chuang–Tzu, Wong–Chongyang, Sun–Tzu, Lie–Yukou and Lao–Tsu (clockwise from the top) sitting around a round table with a bowl of rice in front of them and *a* chopstick placed between each pair of adjacent philosophers. Being Tao monks, they are *silent*, [3] and the only activity they engage themselves in are thinking and eating, alternatively. Each philosopher needs two chopsticks to eat (duh!) and can only use the ones adjacent to them if they are *free*.

[3] reason for their silence is beyond the scope of this tutorial, please refer to the book "Tao is Silent"



Exercise 8 (Dining Philosophers) (4* Points)

Please do the following:

1. Write a CCS statement to model the behaviour of all philosophers and chopsticks.
2. See if there is any deadlock if the philosophers are left to their own devices.
3. Dijkstra's "resource hierarchy solution" ranks each chopstick. A philosopher can only pick a chopstick of lower rank first and then the higher rank if available. Write CCS commands to implement this.
4. Check for deadlocks, liveness and fairness.

□