A Concurrency Workout

Joachim Parrow
Dep. Teleinformatics
KTH, Stockholm, Sweden

October 31, 1996

1 Introduction

The goal of this workout is to make you familiar with a basic language for formulating concurrent processes and its use in formal analysis through an automated tool, the Concurrency Workbench (CWB). The intention is that you should read it while running the CWB and do all examples, exercises and hand-ins in as they come. It will probably take you the better part of a day.

1.1 Prerequisites

- You need to have read chapters 1 and 2.1–2.7 in Milner, *Communication and Concurrency* (Prentice Hall, 1989), or equivalent material, and understood it to an extent that the terms "agent", "action", "transition" etc. are familiar.
- You need to have access to and briefly looked through the manual for the CWB (version 7.0), sections 1-7 and 9, so you know roughly what is in it. You may safely skip everything related to TCCS, SCCS, and divergence. Keep the manual at hand and consult it at need.
- You need to use a machine where CWB version 7.0 is available, and know how to start it. You also need to use a text editor. (No programming is required.)

1.2 Grading

It is strongly recommended that you do all the exercises. The parts labelled "hand-in" (marked $\boxed{\rightarrow}$) serve as checkpoints for grading by teaching assistants. When you solve these keep the following in mind:

- You may complete the hand-ins in **groups of two persons, or individually**, as you wish. No collaboration between groups is allowed.
- The deadline and the possible credits you get for completing the hand-ins in time are defined in the course PM. **The deadline is strict**. If you submit your solution too late, the TAs (depending on their workload) may still grade it and thus give you valuable feedback, but you will receive no credits.
- Your solutions should be **hand-written**. The reason for this is partly to reduce the incentive to collaborate in larger groups, and partly to avoid too long and irrelevant computer listings. Submit **no printouts** from the CWB.
- You must **explain how you have used the CWB** to solve the hand-ins: where appropriate, you should write down the agents under consideration, the CWB commands you apply them do, and brief descriptions and interpretations of the results. This holds even if you feel that a hand-in is simple enough to solve in your head. The point of the hand-ins is to test how well you have mastered the CWB.
- Solutions with unclear diagrams or illegible writing will get a significantly reduced score.

2 Getting Started

2.1 First Steps

Activate the workbench. You are greeted by a message

```
Edinburgh Concurrency Workbench, version 7.0,
Fri Oct  6 11:36:58 BST 1995

Command:
```

The first two lines tell you which version of the Workbench you have. The third line is the CWB command prompt and means that the CWB expects you to type in a command. Let us try the simplest command there is, so type

```
quit
```

which according to the manual terminates execution of the workbench. Unless you read the manual closely you are in for a disappointment. Just typing `quit` (followed by Return) apparently accomplishes nothing! Could it be that the system has crashed? An experienced Unix user will now try to type control-c, in the hope of aborting the program. Try this! What happens?

`quit`

control-c

In the CWB a control-c always returns you to the command prompt. In order to exit from the CWB you need to type the `quit` command properly, i.e., end it with a semicolon:

```
quit;
```

Remember that *all* commands must be terminated by semicolon before the CWB acts on them! Now make sure that you can exit and restart the Workbench!

There is a rudimentary on-line help available through the command `help`. Try

`help`

```
help;
help commands;
help quit;
```

As you see these are not really useful to teach you what you don't know, but they might serve to refresh your memory.

If you (like me) forgot the semicolon again this is a good time to find out that commands may extend over several lines! Typing `help quit;` is equivalent to typing

```
help
quit
;
```

A new line in a command just counts as a blank space. So semicolons are necessary to tell the Workbench when you are finished typing the command.

2.2 Agent Environments

identifiers

You can type in definitions of agents and bind them to *agent identifiers*, sometimes also called *variables* (while in Milner they are called *agent Constants*). This is accomplished with the `agent` command. An example:

agent

```
Command: agent A = a.0;
```

You have now bound the agent identifier `A` to the agent `a.0`; an agent that performs an `a` action and then does nothing more. Agent identifiers must begin with an upper case letter (try to violate this by typing `agent a = a.0;` and check out the error message!) while actions must begin with a lower case letter. Agents identifiers and actions may consist of several characters; the manual will tell you exactly which ones though letters and numerals probably suffice for your purposes now. The agent **0** in Milner is written as `0` (zero).

environment

The bindings of identifiers to agents is stored in what is called the *agent environment*. To see this try the command `pe;` which will print it.

pe

```
Command: pe;

 ** Agents **

agent A = a.0
```

It is permissible to rebind an identifier which is already bound. The old binding is then discarded and replaced by the new one. Try `agent A`

`= a.a.0;` and enjoy the perhaps cryptic warning you get that `A` is already bound, and then check the environment with `pe;`.

Exercise 1 What warning do you get?

clear

The command `clear;` removes all bindings in the environment. You won't really need the binding above again, so why not `clear` it out now?

Exercise 2 Why do you think that the agent identifiers are sometimes referred to as "variables" in the CWB, whereas they are called "constants" in Milner?

If you think that this kind of exercise is unfair, there is a solution in Section 6 below. Most exercises in this workout are solved by using the Workbench. For those who are not you have answers in that section.

2.3 Syntax

Using the `agent` command you can explore the correct syntax for writing agents in the CWB. This is fully defined in the manual and is recapitulated with the `help syntax;` command. But you will only need a subset of it.

actions

Agents are built from actions, which are character strings beginning with a lower case letter. The *co-actions* (overlined in Milner) are written by starting with a single quote (`'`) since overlining is hard to do on a standard keyboard. Thus what in Milner is written $\overline{a}$ is in the CWB written `'a`. And since Greek letters are lacking, the internal action τ in Milner is written `tau` in the CWB[1].

operators

The operators for sum (`+`), parallel (`|`), restriction (\) and relabelling (`[ a / b ]`) are as expected. It is useful to write a couple of agents and check that you know the syntax, and learn from your errors.

Exercise 3 If you find it difficult to commit errors, here are a couple of syntactically incorrect agents which inexperienced users are likely to type. Feed them to the CWB in `agent` commands and digest the error messages. Can you correct the errors, i.e., give syntactically correct agents which probably

[1] In the CWB version 6 and earlier τ was written just `t`.

are what the user intended?

```
a)  a.b.0 | 'a.'b0          b)  (a.b.0 | 'a.'b'.0)\a,b
c)  (a.0+b.0).c.0           d)  a(b.0+c.0)
e)  (a.(b.0|c.0)            f)  B[a\b]
g)  (a.0 + tau.0)\tau       h)  a.(b.0|c.)
```

You may use agent identifiers in agents even if the identifiers are unbound. For example, if `X` is an identifier without a binding it is OK to write `agent A = a.X;`. The CWB will not complain until you try to analyze `A` when `X` is still unbound. You may also use identifiers recursively, as in `agent A = a.A;` and in this way the `agent` command can be thought of as corresponding to Constant definitions, such as $A \stackrel{\text{def}}{=} a.A$, in Milner.

3 Analysis

3.1 Exploring Transitions

`transitions`

The most basic analysis of agent behaviours is to determine transitions. This is accomplished with the command `transitions`, or its equivalent short form `tr`. The command takes one agent as a parameter and will list the transitions from that agent. Examples:

```
Command: transitions a.0;
--- a ---> 0

Command: tr a.E|b.F;
--- a ---> E | b.F
--- b ---> a.E | F
```

Exercise 4 Check the transitions from the following agents (if you wish you may compare with Milner p. 46–47). First try to figure it out in your head, and then use the CWB to verify your effort.

```
a)  a.E | 'a.F              b)  (a.E | 'a.F)\a
c)  (a.E+b.0) | c.F         d)  ((a.0+b.0) | (c.0+d.0))
e)  (a.E+b.0) | 'a.F        f)  ((a.E+b.0) | 'a.F)\a
```

Note that these examples all work when `E` and `F` are unbound.

Exercise 5 Check the transitions from `a.0 + E` when `E` is unbound. What error message do you get? Why don't you get that message for the other examples above? (Partial solution below.)

The `transitions` command only gives the initial transitions from an agent.

```
Command: tr a.E;
--- a ---> E
```

The derivative `E` is not at all analyzed by this command. The CWB does not need to know what `E` is bound to in order to compute the transition. In contrast, when you try `a.0 + E` the workbench will need to know what `E` is bound to, because the transitions from `E` are also transitions from `a.0+E`. So unbound identifiers are not harmful per se, only when their bindings are actually needed to compute the analysis results.

unbound identifiers

The next example is a recursive "a-screamer", something that repeatedly does `a`:

```
Command: agent A = a.A;

Command: tr A;
--- a ---> A
```

Exercise 6 Keep the definition of `A` above and define `agent B = b.B + a.A;`. What are the transitions from `B` here? What are the transitions from `A|B`? Try to figure it out in your head before checking with the CWB.

There is one pitfall when computing transitions from recursively defined identifiers. If the recursion is *unguarded*, i.e., an identifier recurs without any surrounding ("guarding") prefix, the Workbench will not cooperate.

unguarded recursion

```
ommand: agent C = a.0 + C;

Command: tr C;

Resetting tables...


 *** Non-well-founded recursion in C ***
```

In this example `C` occurs both guarded and unguarded, and apparently the unguarded occurrence is enough to make the CWB refuse. The rule of thumb is that if you start with an identifier and can thread you way through definitions to the same identifier without passing a prefix, then that identifier suffers from unguarded (or synonymously "non-well-founded") recursion.

Probably you will often be interested in exploring the behaviour of an agent beyond the initial transitions. Of course you may apply the `tr` command repeatedly to the derivatives to do this. For example, if `A` is bound to `a.b.A` you will find that `A` has one transition leading to `b.A`, and you can apply `tr` again to verify that `b.A` has one transition leading back to `A`. If there are several transitions it becomes cumbersome to repeatedly type in all the agents you encounter in this way. A better way is to use the `sim` command. This command starts a little simulator where you can more conveniently explore transitions. As an example simulate the agent

`sim`

```
(a.b.0 | a.c.0 | 'a.0)\a
```

This agent has three parallel components: two of them begin by `a` and continue with `b` and `c` respectively, and the third begins by `'a` and can therefore synchronize with either of the first two.

```
Command: sim (a.b.0 | a.c.0 | 'a.0)\a;

Simulated agent: (a.b.0 | a.c.0 | 'a.0)\a
Transitions:
    1: --- tau<a> ---> (a.b.0 | c.0 | 0)\a
    2: --- tau<a> ---> (b.0 | a.c.0 | 0)\a
```

```
[0]Sim:
```

This tells us there are two transitions from the agent. Both are τ-transitions arising from a synchronization along `a`. The simulator then prompts us to choose one of these transitions. Let us choose the first by typing 1 followed by semicolon:

```
[0]Sim: 1;

  --- tau<a> --->

Simulated agent: (a.b.0 | c.0 | 0)\a
Transitions:
    1: --- c ---> (a.b.0 | 0 | 0)\a


[1]Sim:
```

This tells us that after the first τ transition the agent can continue with a `c` transition. Again we can choose to explore the behaviour after this transition by typing 1 followed by semicolon. Or we can backtrack to the previous state by typing `return` followed by a number; we will then come back to the numbered position of simulation to make another choice. The simulation position numbers are given in square brackets in front of the simulator prompt. The first one here is 0 so we return to the original agent as follows:[2]

```
[1]Sim: return 0;

Simulated agent: (a.b.0 | a.c.0 | 'a.0)\a
Transitions:
    1: --- tau<a> ---> (a.b.0 | c.0 | 0)\a
    2: --- tau<a> ---> (b.0 | a.c.0 | 0)\a


[0]Sim:
```

[2]In the earliest release of CWB 7.0 there is a bug: "0" (zero) is not recognized as a number in a `return` command. Therefore this example will not work.

In this way you can explore the agent, moving forward and backward through transitions. The simulator has a few other useful commands, such as `history` which gives you a list of the agents encountered so far together with their position numbers. The manual gives a full listing of the commands but initially you will only need the ones mentioned here. You exit the simulator and reach the CWB command prompt through the `quit;` command or through control-c.

Exercise 7 Continue the simulation of the agent above. What are the different agents you can reach? How many transitions are there in all? How does the simulator react when you want to progress from an agent that has no transitions?

3.2 Exploring Deeper

There are many commands in the Workbench which allow you to explore behaviours without stopping at all transitions. One is the `vs` command. `vs`
It computes the "visible sequences" of certain lengths. A visible sequence from an agent is a sequence of visible, i.e. non-τ, actions that the agent can perform. Try the following example of an agent that repeatedly receives signals on `in` and emits them on `out`, until it does an `abort` signal when it stops. To make the example more interesting we let the agent make an internal choice on whether to do the `abort` or continue with the `in out` sequence.

```
Command: agent A = tau.in.'out.A + tau.abort.0;

Command: vs(1,A);
=== abort ===>
=== in ===>

Command: vs(2,A);
=== in 'out ===>

Command: vs(3,A);
=== in 'out abort ===>
```

```
=== in 'out in ===>
```

The `vs` command takes two parameters: an integer signifying the length of the sequences you are interested in, and the agent from which you want to compute the sequences. Note that these must be enclosed in parentheses and separated by a comma — this convention applies to most CWB commands that take more than one parameter.

As is evident from this example the result of the `vs` command does not record the τ-actions. In accordance with the conventions in Milner the double arrow (`==>`) is therefore used to display the output.

Exercise 8 Here `A` can only do the `abort` before `in`, and not between `in` and `out`. Change the definition of `A` so that it can `abort` also between `in` and `out`. Check your solution with the `vs` command.

The `vs` command does not tell you what the derivatives are. Another command, `obs`, will additionally tell you the derivatives (try it on this example). If you want a list of the reachable states from an agent you can get it with the `states` command, and if you only want to know how many different states there are you can use the `size` command. These take just one parameter, the agent to be analyzed. The effect on the example above is:

```
obs
states
size
```

```
Command: states A;
1: 0
2: abort.0
3: 'out.A
4: in.'out.A
5: A

Command: size A;

A has 5 states.
```

Exercise 9 Try `states` and `size` on the agent in Exercise 8.

Unbound identifiers and unguarded recursion are inadmissible for all these commands if the Workbench encounters it while computing the result. An-

other and nastier kind of error is when the state space is infinite. An example of this is in the agent `B = push.(B | pop.0)`.

infinite state space

Exercise 10 Type in this binding and check the visible sequences of length 1 through 4.

Can you see the pattern? This agent can only do `pop` if it is preceded by enough `push`'es — think of `push` as incrementing a counter and `pop` decrementing it, where the counter may never be negative. Are you interested to see how the agent evolves in the transitions?

Exercise 11 Try the simulator on `B` for a few steps until you understand how it works.

The state space of this `B` is infinite — to continue the analogy the counter has one state for each positive integer — so if you try the `states` or `size` command on `B` the CWB will embark on a nonterminating computation trying to get to all states. Eventually the CWB will run out of memory, but you can abort the command with control-c at any point. Unfortunately you get no error message because the CWB cannot know in advance that the state space really is infinite. It is useful to remember that if a command does not appear to terminate the second most common reason is an infinite (or very large) state space. The most common reason, of course, is that you forgot the semicolon.

If the state space is finite but large it may take the CWB some time to complete the command. The following is an educational exercise to find out the performance of your system. Define `C` to be `a.0`. Check that `C` has two states, that `C|C` has four states, that `C|C|C` has eight states etc. By adding on another parallel factor `C` you double the state space, and also the time taken for the CWB to compute it, until you can measure the time with your watch.

Exercise 12 How many states can you do in this way before it takes more than 30 seconds for the CWB to compute them, in the `size` command, on your computer?

Hand-in 1 Let `B = in.'out.B`, i.e., `B` represents a one-place buffer. Let

$$\texttt{C} = (\texttt{B[m/out]}|\texttt{B[m/in]})\backslash\texttt{m}$$

i.e., `C` represents two connected one-place buffers. Draw the transition graph for `C`. A good way is to use the `states` command to learn the reachable states. Write them down well separated on a sheet of paper, and then use the simulator command to explore all transitions between them. Use the transition graph to deduce the visible sequences of length 4 by tracing paths in the graph and then check the result with the `vs` command.

1 →

sort

As a final simple example of analysis commands there is the `sort` command, which calculates the syntactic sort of an agent (Milner Ch. 2.7). This is useful to catch typing mistakes. The `sort` command takes one agent as a parameter and returns the non-restricted action names within it. For example, assume that you have an agent working with actions `a`, `b`, `c` and their co-actions:

```
agent X = (a.b.X | a'.c.0 | 'a.'b.X)\{a,b};
```

Since `a` and `b` are restricted the only name in the sort of `X` should be `c`. But checking this we get

```
Command: sort X;
{a',c}
```

Aha — there is a misprint in that the quote comes on the wrong side of `a`. Easily corrected:

```
Command: agent Y = (a.b.X | 'a.c.0 | 'a.'b.X)\{a,b};
```

That's it isn't it? Perhaps we should check with `sort` again to make real sure:

```
Command: sort Y;
{a',c}
```

What now? Why is the action `a'` still there in the sort?

Exercise 13 Why is it indeed? This is also a common type of error which `sort` often catches!

3.3 File Handling

When you work with larger examples you will probably want to store them in an editable file, to avoid having to retype large amounts of text when you analyze variants of the agents. The easy way here is to use an editor to create a text file containing Workbench commands, exactly as they should be typed interactively to the CWB. Suppose you save this file under the name `myexample.cwb` (the suffix `.cwb` is not mandatory but will help you maintain order among your files). Let that file contain

`input`

```
agent A = a.b.A;
size A;
vs(5,A);
```

You can then execute that file from the CWB by giving it the command `input "myexample.cwb";`. Note that there must be quotes around the filename. The CWB will then execute the commands in that file and display the results. In this case the output is:

```
Command: input "myexample.cwb";

A has 2 states.
=== a b a b a ===>
```

As you see the commands themselves are not displayed. If there is an error in an input file the CWB will revert to receiving input from the terminal. If you have a truly large example you may wish to insert comments into the file. A comment begins with an asterisk , `*`, and extends to the end of the line.

Occasionally a command may produce a large amount of output which you want to go over in an editor. The `output "myoutput.out";` command directs all output from the CWB onto the file `myoutput.out`. This only affects output from successful commands; error messages and such will still be displayed on the terminal. The command `output;` (without a filename) redirects all output to the terminal.

`output`

4 Further Examples

4.1 Resources and Users

In this section we shall consider models of resource managers and users. The resources may be buffer space, processor capacity, communication bandwidth, etc. The users may be programs, protocol entities, pieces of hardware and the like. Users occasionally need resources to complete their tasks, which could be to compute something, to communicate something, to print something etc. For our treatment the particulars do not matter — we could even think of the resources as coffee, beer etc. and the users as students which need these in order to complete their studies. The important thing is the logical procedure for access.

In the simplest case we have a system with one user `U` and one manager `M` controlling one resource. The manager grants access to the resource through the action `'g`. The user `U` repeatedly accesses this resource and performs a task, `t`.

```
agent M   = 'g.M;
agent U   = g.t.U;
agent Sys = (M|U)\g;
```

In this simple case the user should have no problem in always getting the resource. There is no competition — the system works like a bar with only one customer and a bartender who keeps pouring coffee at requests. So the system should behave as something which just repeatedly does `t`.

Exercise 14 Check this with the `vs` command.

Some kinds of design errors will be uncovered by using the `vs` command in this way. For example, if we (mistakenly) had defined a system with a bartender who never does anything (put `M2 = 0` and `Sys2` = `(M2|U)\g` then the inactivity of the system would be manifest already in sequences of length one.

Exercise 15 Check this!

An interesting problem now arises. How long sequences do we actually

need to check with the `vs` command to be absolutely sure that `Sys` does the right thing? This question may appear ridiculous since `Sys` here is so simple. But in principle it is a tough problem. Suppose that `Sys` is something much more complex. How do we know that it will always keep doing these `t` actions?

We can demonstrate the difficulty by defining a similar system with a more erratic bartender who may at any moment resign and leave the customer unattended:

```
agent M3   = 'g.M3 + tau.0;
agent Sys3 = (M3|U)\g;
```

Exercise 16 Check the visible sequences from `Sys3` here.

The visible sequence from `Sys3` are the same as from `Sys`! The reason is that `vs` reports all *possible* sequences, and the bartender `M3` always has the *possibility* of continuing. The fact that the bartender may also resign does not influence the possible sequences. So no matter how long sequences you try you will not see a difference between `Sys` and `Sys3`. On the other hand we would in reality be much happier with a persistent bartender. And if we design a system we would like to ascertain that there is no possibility for the resource managers to suddenly stop.

4.2 More Analysis Methods

We conclude that checking visible sequences is no guarantee of correct behaviour here — we need more refined methods. One such is to simulate the behaviour under study. In this case that is quite effective: when simulating `Sys3` we soon reach a state where the simulator says

`** Deadlocked. **`

meaning there are no further transitions. Since the original intention of the system was to run perpetually such a deadlocked state constitutes an error. However, we may not always be lucky to discover the deadlocks quickly. In general the state space may be so large that interactive simulation is out of the question.

For this reason there are commands in the CWB who perform more or less complex analyzes on agents. An example is the `deadlocks`, or in short form `fd` command. This takes one agent as a parameter and reports all deadlocks, i.e.,. states with no outgoing visible transitions. Try this on `Sys` and `Sys3`:

`deadlocks`
`fd`

```
Command: fd Sys;

No such agents.

Command: fd Sys3;
--- tau ---> (0 | U)\g
```

So there is one deadlock in `Sys3` but no in `Sys` (we also learn through what sequence of actions the deadlock can be reached — sometimes this information is valuable for correcting errors).

Remember that a deadlock is not always a bad thing. Some systems may be intentionally designed to contain terminating states. The CWB will just list the deadlock states but cannot know whether they are intentional terminations or manifestations of design errors — that is your job, as a designer, to figure out! For example, if the user only wants to do 5 transactions `t` we get

```
agent U2   = g.t.g.t.g.t.g.t.g.t.0;
agent Sys4 = (M|U2)\g;
```

and then the system intentionally contains a terminating state which will be found by the `fd` command.

Another example of an analysis command is `eq` which takes two agents as parameters and tells you whether the agents are equivalent. Here "equivalence" is formally defined in Milner Ch. 5 (it is the observation equivalence, also called weak bisimulation equivalence) but for the present we need not be concerned with the precise mathematical definition. Suffice it to say that for two agents to be equivalent their states must "match" in terms of ability to perform observable actions. So whenever one of the agents can do an action, the other can do the same action and again reach a matching state.

`eq`

Now define a `t`-screamer, an agent that does `t` repeatedly, and let that serve as a reference specification of the intended behaviour of the system. We can check which of our systems satisfies this specification:

```
Command: agent Spec = t.Spec;

Command: eq(Sys, Spec);
true
Command: eq(Sys3, Spec);
false
```

The result that `Sys` is equivalent to `Spec` is a quite strong result. `Spec` has only one state, namely itself, so equivalence implies that all states in `Sys` have the same observable behaviour as `Spec`, i.e., can do a `t` action. Of course this does not hold for `Sys3` since there is a deadlock in that agent.

Now `clear` out these definitions and let us consider a resource manager controlling two resources, granting them with `g` and `f`. Let there also be two kinds of users requiring different resources in order to complete different tasks `t` and `u`:

```
agent U1 = f.t.U1;
agent U2 = g.u.U2;
agent M  = 'f.M+'g.M;
agent Sys= (U1|U2|M)\{f,g};
```

We hope that `Sys` will behave as an arbitrary interleaving of `t` and `u` actions — otherwise there is something wrong with our resource manager!

Exercise 17 Check the behaviour of this `Sys` with the `vs` and `fd` command. Find a simple specification which is equivalent to `Sys`.

parameterised agents

It is interesting to explore other resource managers here and determine to what extent they function well with these users. To save ourselves the trouble of typing in the definition of Sys with each such manager we use the CWB facility of *parameterised* agents. This is most easily explained through an example. The parameterised system `Psys`, which depends on a not yet determined resource manager `X`, is

```
agent Psys(X) = (U1|U2|X)\{f,g};
```

The difference from an ordinary `agent` definition is that the identifier to the left of `=` has a formal parameter, here `X`. The agent on the right refers to the formal parameter. Thus `Psys` requires an actual parameter in order to

function as an agent. We can for example write `Psys(M)`, this gives us the same as `Sys`.

```
Command: vs(2,Psys(M));
=== t t ===>
=== t u ===>
=== u t ===>
=== u u ===>
```

We can now check the behaviour of a system with a biased resource allocator that only grants requests from `U1` but refuses to cooperate with `U2`:

```
Command: agent M1 = 'f.M1;

Command: vs(2,Psys(M1));
=== t t ===>
```

Now define other managers:

```
agent M2 = tau.'f.M2 + tau.'g.M2;
agent M3 = 'f.'g.M3;
```

Here `M2` differs from `M` in that it decides on itself whether to grant an `f` or a `g`, and `M3` insists on granting them in order.

Exercise 18 Explore the behaviours of `Psys(M2)` and `Psys(M3)`. Can you correctly guess the visible sequences? Does either of them have a deadlock? How many states do they have? Is either of them equivalent to `Sys`?

If you solved this exercise correctly you may be worried that `Psys(M2)` and `Sys` appear to have the same visible sequences and no deadlocks, yet they are not deemed equivalent. A little experimentation with the simulator reveals that indeed there is a subtle difference between the two: `Psys(M2)` can maneuver, through a sequence of `tau` actions, to a state where `t` is the *only* action available for continuation. In `Sys` on the other hand, whenever `t` is available there appears to be a possibility to continue with `u` (possibly by first doing a `tau` action). In other words, `Psys(M2)` may decide to *refuse* `u`, while `Sys` will not refuse any of `t`, `u`.

Exercise 19 Simulate `Psys(M2)` until you find this state. Simulate `Sys` until you are convinced it has no such state.

blocking environment

This aspect of the behaviour could be important in an environment that does not want to do `t`. We would expect `Sys` to function well in such an environment, while `Psys(M2)` might reach a deadlock if it insists on continuing with `t` and the environment insists on continuing with `u`. We can let the Workbench determine this automatically for us if we analyze agents that represent `Sys` and `Psys(M2)` within such an environment. The simplest such is restricting on `t`, i.e., `Sys\t` corresponds to `Sys` within an environment that blocks `t`, and similarly `Psys(M2)\t` corresponds to `Psys(M2)` within the same environment.

Exercise 20 Analyze these two agents. What is the difference in terms of deadlocks? Does the same idea work if you consider an environment that blocks `u` rather than `t`?

You can also use the equivalence command to get at the difference between `Sys` and `Psys(M2)`. The only action in the sort of `Sys\t` is `u`, so since the manager always grants the appropriate resource we hope that this agent is equivalent to an agent that repeatedly does `u` actions. But `Psys(M2)\t` should not be equivalent if it contains a deadlock, i.e., a state from which no transition is possible.

Exercise 21 Verify this!

You may by now have noticed that equivalence checking is actually faster than deadlock detection on the CWB. This is because more effort has been spent on optimizing the algorithm and the code.

Before proceeding to the next hand-in it may be worthwhile for you to recapitulate the analysis commands mentioned until now. The comments in the right-hand margin are hopefully useful to let you do this quickly!

Hand-in 2 A system containing two users and one manager works as follows. There are two resources, called coffee and sugar. Each user needs both of these and will repeatedly first access coffee and then sugar (always in that order) to accomplish its task. The users do different tasks in this way. In formulas,

$$\mathtt{U} = \mathtt{c.s.u.U} \qquad \mathtt{V} = \mathtt{c.s.v.V}$$

The manager is like `M` above in that it always grants a resource to any requesting user. For the environment of the system it is irrelevant when coffee and sugar are accessed; the only relevant manifestations of the behaviours of U and V are the completion of their tasks, here represented with actions `u` and `v` respectively.

One day the president of the company decided to replace the resource manager with a cheaper model. This cheaper model always grants resources in order; when it has granted coffee it insists on granting sugar before granting coffee again. "Since the manager grants these in the *same* order as both users request it there can be no problem", the president declared. A newly employed engineer was suspicious, however. "What if one user gets coffee", she said, "then might not the system end up in a state where that user *must* complete its task before the second user? This could be significant in an environment which insists that the second user proceeds first." — "This is too complicated for me" the president said. "You have studied the theory of distributed systems recently. Please *prove* to me either that there is a problem with the cheaper manager, if so we will keep the old expensive one, or *prove* that the cheaper manager represents no problem. I don't want to use the expensive one unless it is really necessary."

Luckily the engineer had access to the CWB. What did she do?

The next day it turned out that one of the users, V, had changed behaviour: from now on it takes sugar before coffee! (The other user is as before.) "Surely that can mean no problem for the cheaper manager" the president said. "Suppose the environment desires a `v` action. Then V just waits for U to get coffee. When U has got coffee (but not yet sugar) V can get sugar and coffee, in that order, from the cheaper manager!" But she sent the engineer to redo the analysis just to make sure. What was the result?

2 →

4.3 Semaphores

In a more refined model of resource allocation we consider *critical* resources, i.e., resources that cannot be used by more than one user at a time. A standard way to access such resources are through *semaphores*. For each critical resource there is one semaphore. A user who wants to access the resource "waits" at the semaphore. The semaphore keeps track of waiting users and lets one in at the time. The user "signals" at the semaphore when releasing the resource, so that the semaphore can let in the next user.

If we let "wait" and "signal" constitute actions, a semaphore is simply an agent who alternates these.[3]

```
agent Sem(p,v) = 'p .'v .Sem(p,v);
```

Note in passing that an agent definition can have more than one formal parameter and that these can also be actions. The type of a formal parameter is determined by its first character.

Let there be two users who each performs *two* actions when using a critical resource. `User1` does `a` followed by `b`, and `User2` does `c` followed by `d`.

```
agent User1 = p.a.b.v.User1;
agent User2 = p.c.d.v.User2;
```

Now consider a system of two users and one resource:

```
agent Sys = (User1 | User2 | Sem(p,v))\{p,v};
```

If this works the system should be able to do interleavings of `a b` and `c d`, but nothing can come between `a` and `b` (because at that point one user is at the critical resource, and the other user must therefore wait at the semaphore) or between `c` and `d`.

Exercise 22 Verify this! For example, use the `vs` and `fd` commands, and then try to find an equivalent simple specification equivalent to the system.

A more intersting system consists of two resources and two users. Each user needs *both* resources in order to do its computation:

[3] Dijkstra was the first to treat semaphores extensively. He used "P" for wait actions and "V" for signal actions, in accordance with the Dutch words for these concepts, and this convention has remained in modern computer science.

```
agent User1 = p1.p2.a.b.v2.v1.User1;
agent User2 = p1.p2.c.d.v2.v1.User2;
agent Sys = (User1 | User2 | Sem(p1,v1) | Sem(p2,v2))\{p1,v1,p2,v2};
```

Exercise 23 Verify this example too!

By changing the order in which the users attempt to reserve the resources we get a problem. Change the second `User` so that it first waits on `p2`:

```
agent User2 = p2.p1.c.d.v1.v2.User2;
```

Exercise 24 Show there is a deadlock here!

The deadlock arises, intuitively, if one user obtains one resource and the other user obtains the other resource, and then both users wait for the resource that it lacks. This is the classical deadlock situation, known from operating systems and parallel programming.[4] There are various strategies for breaking up deadlocks (by having users relinquish resources prematurely) or avoiding them (for example by insisting that all users reserve the resources in a particular order).

Hand-in 3 A user is said to be *polite* if whenever it can do a wait operation, it also has the possibility to relinquish (through signal operations) all resources it has reserved. When there are two resources a polite user would thus be defined

```
p1.(p2.a.b.v2.v1.User + v1.User)
```

Now consider the system in the previous exercise where two users reserve resources in different order. Is there a deadlock if both users are polite? Is there a deadlock if one user is normal and one user is polite? Then consider a system with three users and two resources. There are only two ways to order two resources, so with three users two of them must order the resources in the same way. Which of the three users need to be polite for that system to avoid deadlocks?

3 →

[4] The word "deadlock" in computer science normally means just such a situation. The CWB terminology that every state without visible actions is a "deadlock" is not standard.

If we are only interested in detecting this kind of deadlock we do not really need to explicitly represent the `a b c d` actions — these just serve to verify that activities using critical resources cannot be interrupted. In deadlock detection it is enough to let each user signal success.

```
agent User1 = p1.p2.success1.v1.v2.User1;
agent User2 = p2.p1.success2.v2.v1.User2;
```

How much did the state space of `Sys` diminish by this? Perhaps not much, but for larger examples this could become important, and it makes our agents a bit easier to read and understand.

Exercise 25 How about going the whole way and get rid also of the success signals? Why is that a bad idea?

On the other hand we can put the success signals anywhere in the users. We just regard them as indicators that the users progress, so it does not matter if they sit within or outside the critical regions.

Consider next an example with three users and three semaphores:

```
agent User1 = p1.p2.suc1.v2.v1.User1;
agent User2 = p1.p3.suc2.v3.v1.User2;
agent User3 = p3.(p2.suc3.v2.v3.User3 + p1.suc3.v1.v3.User3);
agent Sys   = (User1 | User2 | User3 |
               Sem(p1,v1) | Sem(p2,v2) | Sem(p3,v3))\L;
set   L     = {p1,p2,p3,v1,v2,v3};
```

The third user is a new acquaintance: after having reserved resource number three it is happy with either of resource one or resource two. (In passing we also note the binding of an action set identifier — this makes the agent a bit more readable.)

Does `Sys` here have deadlocks? It has 36 states, so it is now becoming difficult to guess. But we Workbench users don't have to guess!

Exercise 26 Has it a deadlock? What if the third user always insists on the left branch? (Redefine `User3` so that the second summand, after `+`, is

removed.) What if it always insists on the right branch (remove the first summand?)

Even when there are no deadlocks there could still be "partial deadlocks" where a subset, but not all, of the users are blocked waiting for each other. The Workbench will not detect those with the `fd` command as long as there is at least one unimpeded user that can do its success action. Partial deadlocks can instead be detected as follows. Suppose we want to know whether `User1` can be blocked, i.e., if there is a reachable state from which `User1` cannot progress. We then put `Sys` in an environment that hides the other success signals `suc2` and `suc3` from view. This is most easily achieved with the relabelling operator `[tau/suc2,tau/suc3]`. Thus we analyze `Sys[tau/suc2,tau/suc3]`. It has only one observable action, `suc1`, in its sort. If there is no state where `User1` can be permanently blocked, then `Sys[tau/suc2,tau/suc3]` should be equivalent to a `suc1`-screamer.[5]

Exercise 27 Check this, and check if there is a partial deadlock involving any of the other two users. Warning: if you did the previous exercise you first have to reset `User3` to its original definition.

The part of a behaviour obtained by by ignoring (or relabelling to `tau`) actions on some of its ports is sometimes called a *projection* of the behaviour.

projection

Exercise 28 In Exercise 20 we used another kind of environment, a "blocking" environment expressed as restriction; here the "projections" are expressed as relabelling to `tau`. What, intuitively, is the difference between blocking environments and projections and in what different situations are they useful?

Finally, a useful analysis command is `min`, the state space minimization command. In several of the exercises you have been asked to find a specification which is simple enough to be understandable and yet equivalent (using `eq`) to the original system. The CWB can find you this automatically. The command is `min` and it takes two parameters: an agent identifier and an agent. It will analyze the agent, find a minimal (in number of states) agent

`min`

[5] Relabelling to τ is forbidden in Milner but allowed in the CWB.

and bind that to the identifier. It will also tell you how many states there are in the minimal agent. To see the result of the minimization you give the `pe` command. Perhaps surprisingly, `min` is quite efficient — about the same as `eq` — so for large agents the quickest way of finding deadlocks is to minimize them and check (with `pe`) if `0` occurs in their minimized forms. An example, using the definitions above:

```
Command: min(Minsys, Sys);

Minsys has 12 states.
```

Now type `pe` to look at it. It is perhaps too large to be informative. In this example it might be better to instead minimize projections.

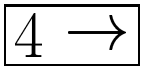
```
Command: min (Min,Sys[tau/suc3]);

Min has 3 states.
```

Exercise 29 Look at these three states using `pe` and convince yourself that this is the expected outcome. Do the same for other projections of `Sys`.

Exercise 30 Consider the system consisting of the following four users and four semaphores. Show that there is no deadlock but a partial deadlock.

```
agent User1 = p1.p2.suc1.v2.v1.User1;
agent User2 = p1.p2.suc2.v2.v1.User2;
agent User3 = p3.p4.suc3.v4.v3.User3;
agent User4 = p4.p3.suc4.v3.v4.User4;
```

Hand-in 4 A surveillance system consists of three parallel processes A, B and C and a set of common critical resources: a data terminal with keyboard, a large video screen, a loudspeaker, a sound detector, a motion detector, and a file system. Processes can reserve and release the resources. A reserved resource can not be reserved by another process until it is released.
Process A runs the interaction with the user. It begins by reserving the the terminal, the screen and the loudspeaker (in that order) and conducts a session with the user. It then releases the terminal and reserves the file system (keeping the screen and speaker) in order to update the files and show a short demonstration. It then releases all resources and starts the procedure from the beginning.
Process B does the surveillance. It reserves the motion detector, the sound detector and the terminal (in that order) and reports any intrusions on the premises. It then releases the motion detector and reserves the speaker, in order to play back any suspicious sounds. It then releases all resources and starts the procedure from the beginning.
Process C runs some background jobs. First it reserves the video screen and the file system, to display the logotype (which is in a huge file!) on the screen. Then it releases the screen but reserves the motion detector, to update some calibration tables. Then it releases the files and reserves the sound detector, to run a correlation check of the detectors. It then releases the detectors and starts the procedure from the beginning.
If a process wants to reserve a resource which is already taken, then it just waits until that resource is released. Can this system deadlock, i.e., can it reach a state where all of A, B and C wait for resources to become released? Can it reach a partial deadlock?



5 Conclusion

Hopefully you now have confidence that these techniques can solve nontrivial problems of naturally occurring kinds. The Workbench supports many more methods. Most notably there is a *modal logic* in which you can state assertions about agents, such as "action a will always precede action b", and have the Workbench check these assertions for you on agents. And there are extensions of the language to represent other aspects of agents like elapsed

real-time. But this is as good a time as any to stop. The Workbench manual contains further material and references to other papers.

6 Solutions to some Exercises

Exercise 2. Here they are called variables because their bindings may vary; an identifier can be rebound to a new agent. For the development of the theory in Milner there is no need for that.

Exercise 3.

Agent	Error	Correction
`a.b.0 | 'a.'b`	missing prefix dot	`a.b.0 | 'a.'b.0`
`(a.b.0 | 'a.'b'.0)\a,b`	missing set brackets	`(a.b.0 | 'a.'b'.0)\{a,b}`
`(a.0+b.0).c.0`	agent before prefix dot	`a.c.0 + b.c.0`
`a(b.0+c.0)`	missing prefix dot	`a.(b.0+c.0)`
`(a.(b.0|c.0)`	missing closing bracket	`(a.(b.0|c.0))`
`B[a\b]`	wrong relabelling slash	`B[a/b]`
`(a.0 + tau.0)\tau`	`tau` cannot be restricted	
`a.(b.0|c.)`	missing trailing `0`	`a.(b.0|c.0)`

Exercise 8. `A = tau.in.(tau.abort.0 + tau.'out.A) + tau.abort.0`

Exercise 13. `Y` is bound to `(a.b.X | 'a.c.0 | 'a.'b.X)\{a,b}` and this agent contains `X` which in turn contains `a'`. Instead `Y` should be bound to `(a.b.Y | 'a.c.0 | 'a.'b.Y)\{a,b}`.

Exercise 17. The specification is `Spec = t.Spec + u.Spec`.

Exercise 22. You might have guessed that the specification is `Spec = a.b.Spec+c.d.Spec` but as you see from `eq` that is wrong. The reason is that if both users wait at the semaphore, one of them will proceed and then the system is in a state where only one of `a b` and `c d` can happen (cf the situation with `Psys(M2)` in the previous section. The correct specification is `Spec = tau.a.b.Spec + tau.c.d.Spec`.

Exercise 25. Without the success signals there are no nonrestricted actions in the system. The sort of the agent thus is the empty set and that means that all its states, formally, are deadlocks: there is no way to continue with observable actions.

Exercise 28. We use a blocking environment, expressed as restriction, to consider the effects of an environment that forces the system under study to execute in a certain way, for example in order to trigger deadlocks. We use projections, expressed as relabelling to `tau`, to disregard some actions the system may make without forcing it to act in one way or another, just to make it easier for us to see what happens when it executes. Thus blocking influences the possible executions of the system and is useful when we suspect we could thereby trigger something interesting, while projection just limits our view of it and is useful when the system is otherwise too complex.