

A Brief Introduction to Mobility Workbench (MWB)

Mikkel Bundgaard

Department of Theoretical Computer Science
IT University of Copenhagen
{mikkelbu}@itu.dk

February 10, 2005

Contents

1	Accessing the Workbench	2
1.1	Downloading and “installing” the Files	2
1.2	Starting MWB	3
2	Syntaxs of CCS Expressions	3
3	Features of the Workbench	3
3.1	Define Agents	4
3.1.1	A simple agent	4
3.1.2	A Two-element Buffer	4
3.2	Running Agents Interactively	5
3.2.1	Running the Two-element Buffer	6
3.3	Handling the Agents	6
3.4	Loading agents or Using Emacs	7
3.5	Using Emacs	7
3.6	Using our Favorite Editor	7
3.7	Deadlock	8
3.8	Quitting MWB	9
3.9	Getting more Information out of MWB	9
4	Reading Error Descriptions in MWB	9
5	Supplementary Reading	10

Introduction

This document contains a brief description of the CCS fragment of the Mobility Workbench (MWB) <http://www.it.uu.se/research/group/mobility/mwb> version MWB'99. MWB is a tool for describing and analysing mobile concurrent systems described in the π -calculus [Mil99, MPW92, Mil91], and *Calculus of Communicating Systems* (CCS) [Mil89].

The document describes the following: how to “install” MWB on either Windows or Linux, how to use MWB, features of MWB, and references for further reading.

1 Accessing the Workbench

You can use a precompiled heap-image, which together with a little bat/shell script (depending on your choice of operating system) should be sufficient to run MWB.

If you want to run MWB at home, you must also have Standard ML of New Jersey (SML/NJ) installed in addition to MWB, since MWB depends on the runtime-environment of SML/NJ. The heap-image has only been tested with SML/NJ version 110.0.7, but it will probably also work with newer versions. SML/NJ can be downloaded from <http://smlnj.org/>.

1.1 Downloading and “installing” the Files

The files necessary for running MWB should be installed at ITU computers (both under Windows and Linux), but if you need to run MWB at home you need the following files, which can be downloaded from the old homepage of the course: <http://www.itu.dk/courses/IMDD/F2004/tools.html> or directly using the following links.

Windows The heap-image for Windows can be downloaded from here¹
<http://www.itu.dk/courses/IMDD/F2004/download/mwb.x86-win32>
and the accompanying bat-file from here
<http://www.itu.dk/courses/IMDD/F2004/download/mwb.bat>.

Linux The heap-image for Linux can be downloaded from here
<http://www.itu.dk/courses/IMDD/F2004/download/mwb.x86-linux>
and the accompanying shell-script from here
<http://www.itu.dk/courses/IMDD/F2004/download/mwb.sh>.

Installing No installation procedure is necessary, since we have a heap-image, just put the two files in the same folder and start MWB by executing the relevant script (executing either `mwb.sh` in a bash-shell or `mwb.bat` in a DOS-prompt). Note that the SML/NJ must be in the PATH in our operating system.

¹Notice that Internet Explorer sometimes removes the extension of the file, so the file must be renamed.

1.2 Starting MWB

When you start MWB then you will be greeted by the following screen².

```

The Mobility Workbench
(MWB'99, version 4.135, built Thu Jan  8 12:20:40 2004)

MWB>
```

The MWB is now ready for an interactive session.

2 Syntax of CCS Expressions

Since this is a introduction to the CCS fragment of MWB, we here present the syntax of a CCS expression P :

$P ::= 0$	the inactive process
$\alpha.P$	perform the action α and continue as P
$P1 \mid P2$	run $P1$ and $P2$ in parallel
$P1 + P2$	run either as $P1$ or $P2$
$Id \langle nlist \rangle$	run as the process named Id instantiated with the names in $nlist$
$(\hat{\ } nlist)P$	restrict the names in $nlist$
(P)	parentheses are used for enforcing precedence

where α can be either **name** (an input action on **name**), **'name** (an output action on **name**)³, or **τ** (the internal action), and **nlist** is a (non-empty) comma-separated list of names. A **name** must be started with a lower-case letter. **Id** is a process identifier, which must start with an upper-case letter.

The parallel operator \mid binds stronger than summation $+$, and both are weaker than prefix $\alpha.P$. So for example the following expression

$$a.b.0 + 'v.0 \mid 'a.'b.0$$

should be read as

$$(a.b.0) + (('v.0) \mid ('a.'b.0))$$

Comments are started with the delimiter $(*$ and ended with $*)$. For examples on the syntax of CCS expressions and what the constructions mean, see Section 3.1 and 3.2.

3 Features of the Workbench

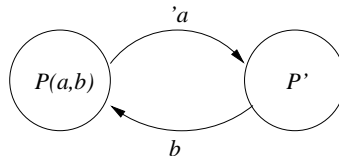
This section describes the relevant features and commands of MWB.

²In all session-snippets mentioned below **MWB>** is written by MWB and the rest of the line is written by the user. All lines not starting with either **MWB>** or **Step>** is written entirely by MWB.

³The sign **'** is the key just to the right of \emptyset on a Danish keyboard.

3.1 Define Agents

3.1.1 A simple agent



An agent declaration of the LTS presented above defines an agent identifier P and can be declared as follows in MWB:

```
MWB>agent P(a,b) = 'a.b.P<a,b>
```

This declares the agent identifier P as a process which first outputs on a , then inputs on b , and then goes back and behave as P again (instantiating with the same names). Processes in MWB must be *closed*, meaning that all unrestricted names used in the process must appear as formal parameters to the process. So since the names a and b are not restricted they appear as formal parameters to the process, otherwise you will get this kind of error.

```
MWB>agent P = 'a.b.P
Error: Definition of P has free names a and b
```

When we use the process we can instantiate these formal parameters with names, otherwise MWB will choose some arbitrary names, which have not been used before. See the first example in Section 3.2, where we instantiate with the names x and y .

When declaring an agent identifier the left-hand side of $=$ must consist of **agent** followed by an agent identifier, again followed by a list of formal parameters (possibly empty). The right-hand side must follow the grammar as defined in the beginning of Section 2.

3.1.2 A Two-element Buffer

If we look at the following declaration of a two-element buffer (accessible from <http://www.itu.dk/courses/IMDD/F2004/material/mwbexamples/buffer.ag>), where we have abstracted away the actual content of the buffer:

```
MWB>agent Buf0(in,out) = in.Buf1<in,out>
MWB>agent Buf1(in,out) = in.Buf2<in,out> + 'out.Buf0<in,out>
MWB>agent Buf2(in,out) = 'out.Buf1<in,out>
```

Here we declare the buffer using three interconnected agent declarations. We use one agent declaration for each *state* that the buffer can be in:

Buf0 The Buffer is empty, so it is only possible to put something in it and it then it behave as **Buf1**.

Buf1 The buffer contains one element, so we can *either* put another element in the buffer (and become **Buf2**), or remove the element from the buffer (and behave as **Buf0**).

Buf2 The buffer now contains two elements (and is thereby full), so the only thing we can do is to remove an element (and continue as **Buf1**).

The buffer can be drawn as the state-transition diagram (or labelled transition system) in Figure 1, where we represent the *states* as circles and the possible actions (*transitions*) of a state is represented by the out-going arcs from the circle.

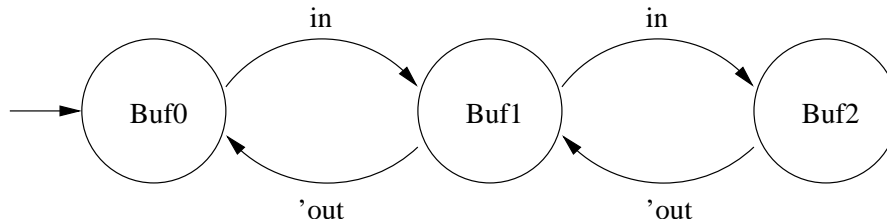


Figure 1: The state-transition diagram of the buffer

Notice that it is necessary to pass the names `in` and `out` as argument between the agents, in order to ensure, that they all use the same names for input and output (and thereby maintain a consistent interface to the environment).

When we later will run or otherwise use the agent definitions, we can instantiate these names to whatever we think is fitting. For a sample run of the buffer see Section 3.2.1.

3.2 Running Agents Interactively

You can use the command `step P` to interactively run a defined process `P` step-by-step. As an example we use the declaration from Section 3.1.1 and we instantiate the names of `P` with `x` and `y`.

At each step (state) of the simulation MWB present us with the different possible actions of the process (numbered from 0 to N). We can then choose one of the actions and MWB will then present us for the new choices etc.

```

MWB>step P<x,y>
* Valid responses are:
  a number N >= 0 to select the Nth commitment,
  <CR> to select commitment 0,
  q to quit.
0: |>'x.y.P<x,y>
Step>0
0: |>y.'x.y.P<x,y>
Step>0
[Circular behaviour detected]
0: |>'x.y.P<x,y>
Step>q
MWB>
  
```

First we instantiate the name `a` with the name `x` and the name `b` with the name `y`. Since the process `P` only has one transition, the workbench only present us this choice:

```

0: |>'x.y.P<x,y>
  
```

We can then type `0` to take this transition (`'x`). We are back at the initial state after we take the second transition, this is indicated by MWB with `[Circular behaviour`

detected]. MWB prints this message every time we enter a state that we have been in before. Typing `q` terminates the simulation.

We can use the command `size P` to get a low measure on the number of states of the process `P`.

```
MWB>size Buf0
3
MWB>size P
2
```

3.2.1 Running the Two-element Buffer

The following is an sample run of the two-element buffer defined in Section 3.1.2.

```
MWB>step Buf0<in, out>
* Valid responses are:
  a number N >= 0 to select the Nth commitment,
  <CR> to select commitment 0,
  q to quit.
0: |>in.(in.Buf2<in,out> + 'out.Buf0<in,out>)
Step>0
0: |>in.'out.Buf1<in,out> (*)
1: |>'out.in.Buf1<in,out> (*)
Step>0
0: |>'out.(in.Buf2<in,out> + 'out.Buf0<in,out>) (**)
Step>0
[Circular behaviour detected]
0: |>in.'out.Buf1<in,out>
1: |>'out.in.Buf1<in,out>
Step>1
[Circular behaviour detected]
0: |>in.(in.Buf2<in,out> + 'out.Buf0<in,out>)
Step>q
...

```

As noted in the previous sections we can decide to instantiate the names to something else (but here we have just decided to keep the original names).

In the concrete example above we first input an element, now we have two possible transitions to choose between (indicated by `(*)` in the example). We choose to input another element. The buffer is now full and the only possible action is to remove an element from the buffer, so only the possibility `0` is available (indicated in the example with `(**)`).

```
0: |>'out.(in.Buf2<in,out> + 'out.Buf0<in,out>)
```

Note that the process that follows the prefix `'out` (the process `in.Buf2<in,out> + 'out.Buf0<in,out>`) is exactly the process expression of `Buf1` as desired.

3.3 Handling the Agents

One can use the command `env` to print all the current agent declarations, and `env Buf0` just to print out the declaration of `Buf0`, if it exists. For example after the

declaration of the two-element buffer in Section 3.1.2 the output is the following:

```
MWB>env
agent Buf0 = (\in,out)in.Buf1<in,out>
agent Buf1 = (\in,out)(in.Buf2<in,out> + 'out.Buf0<in,out>)
agent Buf2 = (\in,out)'out.Buf1<in,out>
MWB>env Buf0
agent Buf0 = (\in,out)in.Buf1<in,out>
```

If one needs to remove an agent declaration then the command `clear P` can be used. `clear` removes all the declarations.

```
MWB>clear Buf0
MWB>env
agent Buf1 = (\in,out)(in.Buf2<in,out> + 'out.Buf0<in,out>)
agent Buf2 = (\in,out)'out.Buf1<in,out>
MWB>clear
Clearing environment.
MWB>env
MWB>
```

3.4 Loading agents or Using Emacs

The basic way of using MWB is to type the agent declarations and commands directly into MWB, this cannot be recommended though. The front-end of MWB does not support many helpful features for text editing: such as copy-and-pasting, retyping the last command, and editing an old command. So, if you for example type a long command

```
agent P(x,y) = 'x.y. ...
```

and then press enter, just to receive `Error in line 1: syntax error found at EOL`, then you have to type the entire command again to correct the error, and this will soon be quite annoying ;-). Therefore it is recommended to use one of the following methods:

3.5 Using Emacs

This method describes how one can use `Emacs` for typing commands in MWB. Since `Emacs` is quite different compared to “traditional” editors, one should only choose this solution if they are either familiar with `Emacs` or are willing to spend some (read: a lot of) time getting to know it.

`Emacs` contains a useful feature for being a front-end for interactive sessions (as found in different kinds of programming languages like e.g. *ML* and *Prolog*). Using the command `M-x shell` one can run a shell or DOS-prompt through `Emacs`. This has the advantage that one has all the features of `Emacs` available together with a standard shell. E.g. going back through the history of the last 30 commands with `M-p`, forward with `M-n`, and the usual copy-paste shortcuts etc.

3.6 Using our Favorite Editor

If you are not comfortable with `Emacs`, then you can instead use the favorite editor of our choice. In this editor you then type your agent definitions. When you are done

typing your agent definitions, you can then use the command `input "filename"` to load the definitions into MWB⁴. If you for example have a text file `agents.ag` (accessible from <http://www.itu.dk/courses/IMDD/F2004/material/mwbexamples/agents.ag>) containing agent definitions of agent P and agent Q:

```
agent P(x,y) = 'x.Q<x,y>
agent Q(x,y) = y.P<x,y>
```

and a fresh MWB session

```
MWB>
```

then you can load these definitions into your MWB session by using the `input` command.

```
MWB>input "agents.ag"
```

MWB only responds if an error occurs, e.g. if the file `agents.ag` does not exist or if it contains errors in the code. So to verify, that our definitions have been loaded correctly we use the command `env` to print all agent definitions.

```
MWB>env
agent P = (\x,y)'x.Q<x,y>
agent Q = (\x,y)y.P<x,y>
MWB>
```

3.7 Deadlock

Using the command `deadlocks` we can check if an agent can deadlock or not. In this setting an agent can deadlock if the its denotation (LTS) contains a state that has no out-going transitions. Looking back on our definition of the two-elements buffer from Section 3.1.2 we tests if it can deadlock or not.

```
MWB>deadlocks Buf0<in,out>
No deadlocks found.
```

So MWB finds out that it is not possible for the buffer to deadlock. If we change the definition of the last state of the buffer from

```
MWB>agent Buf2(in,out) = out.Buf1<in,out>
```

to:

```
MWB>agent Buf2(in,out) = in.0 + out.Buf1<in,out>
```

which represent that the buffer throws an exception (or fails), if one tries to put an element into it even though it is full. Now MWB detects that the buffer can deadlock:

```
MWB>deadlocks Buf0<in,out>
Deadlock found in 0
reachable by 3 commitments
```

⁴Actually you can also type some of the interactive commands in the text file, but normally this is not preferred.

In order to get some more information about the possibility of deadlock we can increase the debugging level of MWB.

```
MWB>set debug 1
MWB>deadlocks Buf0<in,out>
Deadlock found in 0
  reachable by 3 commitments
  |>in(1)|>in(1)|>in(1)
```

Now the command `deadlocks` tells us that the agent deadlock (from initial state `Buf0`) after three consecutive inputs (the *trace* `|>in(1)|>in(1)|>in(1)` show us this). We can also check this ourself using the `step` command.

```
MWB>step Buf0<in,out>
* Valid responses are:
  a number N >= 0 to select the Nth commitment,
  <CR> to select commitment 0,
  q to quit.
0: |>in.(in.Buf2<in,out> + out.Buf0<in,out>)
Step>0
0: |>in.(in.0 + out.Buf1<in,out>)
1: |>out.in.Buf1<in,out>
Step>0
0: |>in.0
1: |>out.(in.Buf2<in,out> + out.Buf0<in,out>)
Step>0
No commitments for 0
Quitting.
```

The run show us after three consecutive inputs (from the initial state), we end up in a state from which there are no possible actions. See Section 3.9 for more information about the `set debug` command.

3.8 Quitting MWB

You can exit from MWB using the command `quit`.

3.9 Getting more Information out of MWB

Using the `set debug n`, where n is a non-negative integer one can increase/decrease the amount of information from MWB. The default setting is 0 (meaning no additional information), when one needs more information from MWB, then 1 is probably the most appropriate setting. For values above 1 the information often becomes too overwhelming, and thereby useless.

4 Reading Error Descriptions in MWB

The error description of syntax errors in MWB is unfortunately not always precise. In the example below we have forgotten to make `y` a part of the definition of `P`.

```
MWB>agent P(x) = 'x.y.0
Error: Definition of P has free name y
```

In the example below we have forgotten to end the processes with a trailing 0, but the error description only tells us that the error is near the token PAR (the symbol |)⁵.

```
MWB>agent P(x) = 'x | x
Error: syntax error found at PAR
```

```
MWB>agent P(x) = 'x.0 | x.0
MWB>
```

As can be observed, the error description does not use the concrete syntax, but instead uses the names of the tokens. A translation between the most common token names and their actual syntax is provided in Table 1.

Syntax	Name of Token
	PAR
Name of Actions x, y, \dots	ACT
(LPAR
)	RPAR
.	DOT
0	NIL
=	EQUALS
t	TAU
>	GREATERTHAN
<	LESSTHAN
1	ONE
End of Line	EOL

Table 1: Syntax and Tokens

5 Supplementary Reading

The following references, which can be found on the homepage under “Tools, notes, etc”, are not part of the curriculum, but serves as a good foundation for understanding MWB. For texts describing CCS and π -calculus see the references in the Introduction.

- [Vic95] is the main material for this document. The manual contains information about additional commands, the model checker and its associated logic, but is quite succinct in its description of the syntax and the basic commands.
- [VM94] the paper about the original prototype of MWB. The paper contains a brief introduction to π -calculus, the implementation of equality checking, and some example sessions.
- [Vic94] a more thorough description of the concepts presented in the User Guide.

⁵This is a result of MWB uses the standard lexer- and parser-generators of SML/NJ.

- [Bes98] A Master's thesis describing the model checker and logic in the new version of MWB (MWB'99). The thesis also contains descriptions of: λ -calculus, π -calculus, and the implementation of the model checker.

References

- [Bes98] Fredrick B. Beste. The model prover — a sequent-calculus based modal μ -calculus model checker tool for finite control π -calculus agents. Master's thesis, Department of Computer Systems, Uppsala University, 1998. Available as report DoCS 98/97.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [Mil91] Robin Milner. The polyadic π -calculus: A tutorial. Technical Report ECS-LFCS-91-180, LFCS, Department of Computer Science, University of Edinburgh, 1991.
- [Mil99] Robin Milner. *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, 1999.
- [MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Journal of Information and Computation*, 100:1–40 and 41–77, 1992.
- [Vic94] Björn Victor. *A Verification Tool for the Polyadic π -Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, 1994. Available as report DoCS 94/50.
- [Vic95] Björn Victor. *The Mobility Workbench User's Guide: Polyadic version 3.122*. Department of Information Technology, Uppsala University, 1995.
- [VM94] Björn Victor and Faron Moller. The mobility workbench — a tool for the π -calculus. Technical Report DoCS 94/45, Department of Computer Systems, Uppsala University, 1994. Also available as Technical Report ECS-LFCS-94-285, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.