The Mobility Workbench User's Guide
Polyadic version 3.122

Björn Victor

October 9, 1995

Contents

1 Introduction

The Mobility Workbench (MWB) is a tool for manipulating and analyzing mobile concurrent systems described in the π-calculus [MPW92, Mil91], developed by Björn Victor[1], Faron Moller[2], Lars-Henrik Eriksson[3] and Mads Dam[4]. It is written in Standard ML, and currently runs under the New Jersey SML compiler.

In the current version, the two basic functionalities are equivalence checking and model checking.

The tool implements algorithms [Vic94] to decide the *open bisimulation equivalences* of Sangiorgi [San93], for agents in the polyadic π-calculus with the original positive match operator. This is decidable for π-calculus agents with *finite control*, corresponding to CCS finite-state agents, which do not admit parallel composition within recursively defined agents.

The algorithm is based on the alternative "efficient" characterizations of the equivalences described in [San93, Vic94], and generates the state space "on the fly". Algorithms for both the strong and weak equivalences are implemented.

The tool also contains an experimental implementation of Mads Dam's model checker [Dam93].

There are also commands e.g. for finding deadlocks and interactively simulating an agent.

We refer to [MPW92, Mil91, San93, Vic94, Dam93] for the formal framework of the tool; the π-calculus, the definition of the equivalences, the modal logic, etc.

The MWB is undergoing constant and dynamic changes. This guide describes the current version as of October 1995. Some parts of the guide will be rewritten, and a section on sortings will be added.

2 Input syntax

Input lines can be split using the continuation character "\" at the end of an input line, or (perhaps preferrably) by wrapping things in parentheses. Anything between "`(*`" and "`*)`" is a comment and is treated as whitespace. Note that comments cannot (at present) be nested.

The syntax of agents is given by the following grammar:

$$\begin{array}{rcl} P & ::= & \mathtt{0} \;\Big|\; \alpha.P \;\Big|\; \mathit{pfx}.P \;\Big|\; [a=b]P \;\Big|\; P_1 \mid P_2 \;\Big|\; P_1 + P_2 \;\Big| \\ & & \mathit{Id}{<}\mathit{nlist}{>} \;\Big|\; (\hat{}\,\mathit{nlist})P \;\Big|\; (\backslash \mathit{nlist})P \;\Big|\; [\mathit{nlist}]P \;\Big|\; (P) \end{array}$$

where *nlist* is a (non-empty) comma-separated list of names; α is an action: τ (silent) or a name (input) or a co-name (output); *pfx* is an abbreviated prefix (see below); and *Id* is a name starting with an upper-case letter. Names must start with a lowercase letter but can after that include the characters _, $, ', letters and digits. The parallel operator | binds stronger than summation +. Both bind weaker than prefix . and match [...].

[1]Department of Computer Systems, Uppsala University, Box 325, S-751 05 Uppsala, Sweden; email: `Bjorn.Victor@DoCS.UU.SE`. Work supported by the ESPRIT BRA project 6454 "CONFER"

[2]SICS, Box 1263, S-164 28 Kista, Sweden; email: `fm@sics.se`. Work supported by the ESPRIT BRA projects 7166 "CONCUR2" and 6454 "CONFER".

[3]Logikkonsult NP AB, Swedenborgsgatan 2, S-118 48 Stockholm, Sweden; email: `lhe@lk.se`. Work supported by the ESPRIT BRA projects 6454 "CONFER" and 8130 "LOMAPS".

[4]SICS, Box 1263, S-164 28 Kista, Sweden; email: `mfd@sics.se`. Work supported by the ESPRIT BRA project 8130 "LOMAPS".

F	::=	`TT`	Truth
		`FF`	Falsity
		$a = b$	Equality between names
		$a\#b$	Inequalty between names
		$F_1\&F_2$	Conjunction
		$F_1 \| F_2$	Disjunction
		`not` F	Negation
		$<\alpha>F$	Possibility modality
		$[\alpha]F$	Necessity modality
		`Sigma` $a.F$	Sigma-expression
		`Bsigma` $a.F$	Bound sigma
		`Pi` $a.F$	Universal quantification
		`exists` $a.F$	Existential quantification
		$Id(nlist)$	Use of fixpoint identifier
		$(\sigma Id(nlist).F)(nlist)$	Fixpoint expression
		$\sigma Id.F$	Alternative to the above without args
		(F)	

where σ is either `mu` (least fixpoint operator) or `nu` (greatest fixpoint operator).

Figure 1: Syntax of formulae

The following translations and shorthands are used:

Input	Translation	
`^`	$\boldsymbol{\nu}$	*restriction*
`\`	λ	*abstraction*
`0`	$\mathbf{0}$	*null process*
`'`α	$\overline{\alpha}$	*output action*
`t`	τ	*internal action*
`a(`*nlist*`).`	`a.(\`*nlist*`)`	*input prefix*
`'a<`*nlist*`>.`	`'a.[`*nlist*`]`	*output prefix*

2.1 Model checking

The syntax of formulae is given by the grammar in Figure 1.

A brief description of the semantics is given in Figure 2. For full details, please refer to [Dam93].

Note that modalities *bind* the action. That is, given a formula such as $<x>P$, x is *bound* in P to the name of the action of some transition the agent can perform. Example: $a.A + b.B \models <x>P$ iff $A \models P\{a/x\}$ or $B \models P\{b/x\}$. Another example: $a.A + b.B \models [x]P$ iff $A \models P\{a/x\}$ and $B \models P\{b/x\}$.

Modal logics often use another semantics where the *actual* name of the action is inside the diamond or box, rather than a bound variable. To achieve the same effect with our semantics, write:

Other semantics	Our semantics
$[a]P$	$[x](a\#x \| P)$
$<a>P$	$<x>(a = x\&P)$

Note also that, because of implementation issues, fixpoint formulae must be closed. E.g. `nu D.<x>(x=b&D)` is invalid, but the equivalent formula `(nu D(b).<x>(x=b&D(b)))(b)` is OK. This will be remedies in the near future.

$A \models \texttt{TT}$	Always true.
$A \models \texttt{FF}$	Always false.
$A \models a = b$	True iff a and b are the same names.
$A \models a \# b$	True iff a and b are different names.
$A \models P \& Q$	True iff $A \models P$ and $A \models Q$.
$A \models P \vert Q$	True iff $A \models P$ or $A \models Q$.
$A \models \texttt{not}\ P$	True iff not $A \models P$.
$A \models <x>P$	True iff the agent can commit to some input action $A \succ a.A'$ and $A' \models P\{a/x\}$.
$A \models <'x>P$	True iff the agent can commit to some output action $A \succ' a.A'$ and $A' \models P\{a/x\}$.
$A \models [x]P$	True iff for every input commitment $A \succ a.A'$ the agent can perform, $A' \models P\{a/x\}$.
$A \models ['x]P$	True iff for every output commitment $A \succ' a.A'$ the agent can perform, $A' \models P\{a/x\}$.
$[a]A \models \texttt{Sigma}\ x.P$	True iff $A \models P\{a/x\}$.
$(\hat{}y)[y]A \models \texttt{Bsigma}\ x.P$	True iff $A\{a/y\} \models P\{a/x\}$, where a is a new name.[5]
$A \models (\sigma D(x_1, \ldots, x_n).P)(a_1, \ldots, a_n)$	Fixpoint formula. True iff the appropriate fixpoint of P is true. σ should be `nu` for the greatest fixpoint or `mu` for the least fixpoint. The fixpoint is a predicate with formal arguments $x_1, \ldots, x_n$ and actual arguments $a_1, \ldots, a_n$. Within P, D is bound to the fixpoint expression itself.

Figure 2: Brief semantics of formulae.

3 Commands of the MWB

3.1 `help`

gives a general help text. `?` (questionmark) is a synonym for this command.

3.1.1 `help` *command*

gives a help text for *command*.

3.2 `quit`

terminates the program. End-of-file (typically `Control-D`) is a synonym for this command.

3.3 `agent`

defines an agent identifier. Two equivalent examples:

```
agent P(x,y) = (^z)'x<y,z>.y(x,y).P<y,x>
agent P = (\x,y)(^z)'x.[y,z]y.(\x,y)P<y,x>
```

An agent definition must be closed, i.e., its free names must be a subset of the argument list. Only guarded recursion is handled.

3.4 `clear`

removes agent identifier definitions. `clear P` removes the definition of the agent identifier P, while `clear` without an argument removes all definitions.

3.5 `env`

prints all agent definitions in the environment. `env P` shows the definition of the agent identifier `P`.

3.6 `input "`*filename*`"`

reads commands from the file named *filename*. The double quotes are part of the syntax but not of the filename.

3.7 `eq` $agent_1$ $agent_2$

checks whether $agent_1$ and $agent_2$ are strong open bisimulation equivalent.

If the two agents *are* equivalent, a bisimulation relation is available[6] for inspection by the user.

3.8 `eqd` `(`$name_1$,...,$name_n$`)` $agent_1$ $agent_2$

checks whether $agent_1$ and $agent_2$ are strong open bisimulation equivalent given the distinction formed by making $name_1$, ..., $name_n$ distinct from all free names in $agent_1$ and $agent_2$. $\{name_1, \ldots, name_n\}$ should be a subset of the free names of $agent_1$ and $agent_2$. (Names not free in $agent_1$ or $agent_2$ are meaningless and are simply removed).

[6] if MWB is running interactively, i.e. not reading commands from a file.

3.9 `weq` $agent_1$ $agent_2$

checks whether $agent_1$ and $agent_2$ are *weak* open bisimulation equivalent.

3.10 `weqd` ($name_1$,...,$name_n$) $agent_1$ $agent_2$

checks whether $agent_1$ and $agent_2$ are weak open bisimulation equivalent given the distinction formed by making $name_1$,..., $name_n$ distinct from all free names in $agent_1$ and $agent_2$. $\{name_1, \ldots, name_n\}$ should be a subset of the free names of $agent_1$ and $agent_2$.

3.11 `check` *agent formula*

Responds `yes` if the *agent* is a model for the *formula*, otherwise `no`.

3.12 `sort` *agent*

Displays the object sort and most general sorting of *agent*, or gives an error message if the *agent* doesn't respect any sorting.

3.13 `deadlocks` *agent*

finds and describes deadlocks in the agent given as argument. It displays the agent in which the deadlock is found.

The deadlocks are displayed as they are found, which makes the command useful even if the state space is infinite.

3.14 `step` *agent*

interactively simulates the agent, by presenting the possible commitments of the agent and letting the user select one, and repeating this until there are no possible commitments. Typing `q` terminates the simulation.

3.15 `size` *agent*

gives a low measure of the graph size of the agent. This is not always minimal, but the agent space being explored by the equivalence checking commands is possibly larger.

3.16 `time` *command*

performs the command[7] and prints timing information for its execution.

3.17 `set`

sets various parameters of the MWB. `set ?` shows what can be set.

3.17.1 `set debug` n

sets the debugging level of the program. n should be a non-negative integer; the only value we expect to be valuable to users other than the developers is 0 (meaning debugging is turned off). The use of this command for higher values of n is discouraged, and as such is left undocumented here.

[7] in non-interactive mode

3.17.2 `set threshold` n

sets the rehashing threshold of the internal hashtables to $n\%$. n should be between 1 and 100; its inital value is 30.

3.17.3 `set remember` *on/off*

sets whether commitments are recorded in hashtables whenever they are computed, so as to save computational work. For large agents, this may require large amounts of memory. Using `set remember off` lowers the memory requirements, but may instead increase the runtime.

3.17.4 `set rewrite` *on/off*

sets the automatic rewrite flag on or off. With rewriting on, $(\boldsymbol{\nu} x)P \Rightarrow \mathbf{0}$ if $\forall \alpha : P \succ \alpha.P', \mathrm{n}(\alpha) = x$. Since the commitments of P are computed to see if the rewrite is applicable, we do *not* recommend using `set rewrite on` in combination with `set remember off`. With `set remember on` however, there is no extra cost for computing these commitments.

3.18 `show`

shows various parameters of the MWB. `show ?` shows what can be shown.

3.18.1 `show debug`

shows the debug level.

3.18.2 `show threshold`

shows the rehash threshold.

3.18.3 `show remember`

shows the remember setting.

3.18.4 `show version`

shows the version of the MWB.

3.18.5 `show all`

shows all of the above.

3.18.6 `show tables`

shows the sizes etc of the internal hash tables used for recording commitments.

4 Example use

In Figure 3 we have a sample session which demonstrates some simple usage.

In the sample session, we first define an agent `Buf1` implementing a one-place buffer, then another, `Buf2`, implementing a two-place buffer by composing two instances of `Buf1`, and finally three agents, `Buf20`, `Buf21` and `Buf22`, together implementing a two-place buffer without parallel composition.

```
The Mobility Workbench
(Polyadic version 3.122)

MWB> agent Buf1(i,o) = i(x).'o<x>.Buf1(i,o)
MWB> agent Buf2(i,o) = (^m)(Buf1(i,m) | Buf1(m,o))
MWB> agent Buf20(i,o) = i(x).Buf21(i,o,x)
MWB> agent Buf21(i,o,x) = i(y).Buf22(i,o,x,y) + 'o<x>.Buf20(i,o)
MWB> agent Buf22(i,o,x,y) = 'o<x>.Buf21(i,o,y)

MWB> weq Buf2(i,o) Buf20(i,o)
The two agents are related.
Relation size = 18.  Do you want to see it?  (y or n) y
R= < (^v2)(i.(\x)'v2.[x]Buf1<i,v2> | v2.(\x)'i.[x]Buf1<v2,i>),
      i.(\x)Buf21<i,i,x> >  {}
      . . .

MWB> step Buf2(i,o)
   0:  |>i.(\v2)(^v3)('v3.[v2]Buf1<i,v3> | v3.(\x)'o.[x]Buf1<v3,o>)
Step> 0
  Abstraction (\v2)
   0:  |>t.(^v3)(i.(\x)'v3.[x]Buf1<i,v3> | 'o.[v2]Buf1<v3,o>)
Step> 0
   0:  |[i=o]>t.(^v3)('v3.[v2]Buf1<i,v3> | v3.(\x)'o.[x]Buf1<v3,o>)
   1:  |>i.(\v3)(^v4)('v4.[v3]Buf1<i,v4> | 'o.[v2]Buf1<v4,o>)
   2:  |>'o.[v2](^v3)(i.(\x)'v3.[x]Buf1<i,v3> | v3.(\x)'o.[x]Buf1<v3,o>)
Step> 1
  Abstraction (\v3)
   0:  |>'o.[v2](^v4)('v4.[v3]Buf1<i,v4> | v4.(\x)'o.[x]Buf1<v4,o>)
Step> quit

MWB> agent Buf22 = (\i,o,x,y)('o.[x]Buf21(i,o,y) + [i=o]t.0)

MWB> weq Buf2 Buf20
The two agents are NOT related.

MWB> weqd (i) Buf2(i,o) Buf20(i,o)
The two agents are related.
Relation size = 8.  Do you want to see it?  (y or n) y
R = < (^v2)(i.(\x)'v2.[x]Buf1<i,v2> | v2.(\x)'o.[x]Buf1<v2,o>),
      i.(\x)Buf21<i,o,x> >  {i#o}
      . . .
```

Figure 3: A simple sample session with the MWB.

We proceed with this example by comparing the two implementations for weak equality. The MWB responds by saying that they are equivalent and that it found a bisimulation relation with 18 tuples, and asks us if we want to inspect it. We respond positively and the MWB prints out the relation as a list of pairs of agents with associated distinction sets.

We then simulate the behaviour of the agent `Buf2(i,o)`. The MWB presents the possible commitments, including their least necessary conditions (if not trivial), and prompts the user to select one of them. When the user selects a commitment whose derivative is an abstraction or concretion, the bound names are instantiated automatically. After having a single choice on the first two steps, we then get a choice of three commitments; the first which is possible only if the names `i` and `o` are the same.

Next, we change the definition of `Buf22` to introduce a possible deadlock and again check for weak equivalence between `Buf2` and `Buf20`, this time as abstractions, without instantiating their arguments. We find that they are not equivalent, and proceed by trying to equate `Buf2(i,o)` and `Buf20(i,o)` under the proviso that `i` is different from all other free names of the two agents (namely `o`). Under this distinction, there are no deadlocks, and the MWB reports that they are once again equivalent.

5 Availability

The MWB is available by anonymous FTP from the host `ftp.docs.uu.se` in the directory `pub/mwb`. The file `README` contains further directions and information. An up-to-date version of this guide is always part of the distribution.

Binary executables are provided for some architectures and operating systems. Source code is also provided which can be compiled with the SML-NJ compiler. SML-NJ is currently available from the host `ftp.research.att.com`, directory `dist/ml` and the host `princeton.edu`, directory `pub/ml`.

There is also information on the MWB available on the World Wide Web, in the URL `http://www.docs.uu.se/~victor/mwb.html`.

Any bug reports, queries, feedback etc should be sent to:

email: `mwb-bugs@DoCS.UU.SE`
fax: +46 18 550225
mail: Björn Victor
Dept. of Computer Systems
Uppsala University
Box 325
S-751 05 Uppsala
SWEDEN

References

[Dam93] M. Dam. Model checking mobile processes. In E. Best, editor, *CONCUR'93, 4^{th} Intl. Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 22–36. Springer-Verlag, 1993. Full version in Research Report R94:01, Swedish Institute of Computer Science, Kista, Sweden.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

[MPW92] R. Milner, J. Parrow and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.

[San93] D. Sangiorgi. A theory of bisimulation for the π-calculus. Technical Report ECS-LFCS-93-270, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, June 1993. A revised version will appear in Acta Informatica. An extended abstract appeared in E. Best, editor, *CONCUR'93, 4^{th} Intl. Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 127-142. Springer-Verlag, 1993.

[Vic94] B. Victor. *A Verification Tool for the Polyadic π-Calculus.* Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.