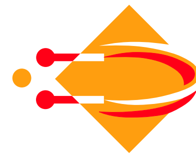

An Implementation of the MR Calculus

Theory Department, IT University of Copenhagen

Jens Christian Godskesen (jcg@it.edu)
Thomas Hildebrandt (hilde@it.edu)
Søren Eduard Jacobsen (sej@it.edu)

IT University Technical Report Series **TR–TR-2002-14**

ISBN 87-7949-017-4

Copyright © , Jens Christian Godskesen (jcg@it.edu)
Thomas Hildebrandt (hilde@it.edu)
Søren Eduard Jacobsen (sej@it.edu)

IT University of Copenhagen
All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

ISSN 1600–6100

ISBN 87-7949-017-4

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone:	**+45 38 16 88 88**
Telefax:	**+45 38 16 88 99**
Web	`www.it-c.dk`

Abstract

We demonstrate a simulator for the Mobile Resource Calculus (MR), called **MRsim**. First, an overview of the syntax and semantics of MR is given, along with an explanation of how the calculus is implemented. The second part of the paper gives an example of how the simulator is used.

1 Introduction

This paper describes an implementation of a mobile calculus called the "Calculus of Mobile Resources" (MR), as described in [1]. Actually, only a subcalculus is implemented where slots cannot be deleted. Another difference is that the paths referring to slots are reversed. The program - written in Standard ML - is called **MRsim**, for Mobile Resource Simulator. The paper also gives an overview of the syntax in order to justify the datatype design, but is not meant to give a full introduction to MR. For details on this, see [1]. The source code is included in the appendix. It is recommended to have **MRsim** running while reading this document.

1.1 An Example Session

To run MRsim, you must have a Standard ML compiler/interpreter installed. MRsim has been written using Moscow ML[1], but any Standard ML implementation should do. This example is only meant to give a feel for the `MRsim` system, the interesting features of MR (e.g. mobility) are introduced later in this paper. Start up an interactive session and apply the file "mr.sml" by typing:

```
- use "mr.sml";
```

This will load the datatypes and functions needed to define MR expressions, pretty-print them, and run the transition functions. There are also a number of examples available (given in the file "examples.sml"). The pre-defined variable `e1` holds the expression $a.0 \mid \bar{a}.0$. If you type

```
- e1;
```

at the prompt, you get:

```
> val it =
    PAR(PREFIX(ACTION(ACNAME "a"), NIL), PREFIX(ACTION(coACNAME "a"), NIL)) :
  prex
```

which is an abstract representation of the data, see section 1.2 for more detail on this. For a more human-readable form, try:

```
- pp e1;
```

which will yield

```
> val it = "a.0|'a.0" : string
```

Notice that the prefixed "`'`" means co-action. There are several pretty-printing functions available, see section A for details. The transition function `t` will return a tuple list of possible actions paired with the resulting expressions:

```
- t e1;
> val it =
    [(LAMBDA(ACTION TAU), NIL),
     (LAMBDA(ACTION(coACNAME "a")), PREFIX(ACTION(ACNAME "a"), NIL)),
     (LAMBDA(ACTION(ACNAME "a")), PREFIX(ACTION(coACNAME "a"), NIL))] :
  (pi * prex) list
```

To render the output more readable, use the pretty-print function `ppal` on the output:

```
- ppal it;
(tau, 0)
('a, a.0)
(a, 'a.0)
> val it = () : unit
```

Where `it` is used to reference the last output by the command-line interpreter.

[1] Obtainable from http://www.dina.kvl.dk/~sestoft/mosml.html

1.2 Syntax and Datatypes

The sets $\mathcal{P}$ of *process expressions* is defined by:

$$p, q \ ::= \ 0 \mid n\lfloor r \rfloor \mid \lambda.p \mid p \parallel q \mid !p \mid (n)p \quad (\mathcal{P}) \tag{1}$$

$$r \ ::= \ \bullet \mid p \tag{2}$$

In Mrsim, this is implemented as follows:

```
datatype prex =
  NIL                        (*Empty Constructor              *)
| SLOT of rname * res        (*SLOT is a resource (slot)      *)
| PREFIX of lambda * prex    (*prefix 'action' constructor    *)
| PAR of prex * prex         (*PAR parallel (||) constructor  *)
| REP of prex                (*Replication (!)                *)
| REST of rest * prex        (*Restriction                    *)
| PROCESS of string          (*PROCESS (atomic) constructor   *)

and res =
  EMPTY                      (*slots can be empty, ready to   *)
| RESOURCE of prex           (*accept a resource.             *)
```

The prefix λ is defined by:

$$\lambda \ ::= \ \alpha \mid \rho \tag{3}$$

$$\rho \ ::= \ \alpha\delta \mid \delta \triangleright \overline{\delta'} \tag{4}$$

which define the set $\mathcal{L}$ of *labels*. The actions α play the same role as in CCS [2], and δ is a *direction path*. Labels are implemented in conjunction with the above:

```
and lambda =
  ACTION  of action           (*ACTION is the action prefix   *)
| DIRAC of lambda * addr      (*Directed action               *)
| MOVE  of addr * addr        (*MOVE will be infixed '>'      *)
and action =
  ACNAME of string
| coACNAME of string
| TAU
```

The datatypes for the MR grammar are the following: We have a datatype *prex* that models the grammar in 1 and 2, with constructors for parallel, replication, etc. and a datatype *lambda* that models the labels in 3 and 4. Note that it is possible to name processes with the PROCESS constructor, though it is not defined in the MR grammar. The lambda datatype has a "helper" datatype called action, defined by:

$$action ::= a \mid \overline{a} \mid \tau \tag{5}$$

The grammar in 5 is for distinguishing between actions and co-actions, τ is the silent action. This representation yields an abstract syntax, where the process expression

$$a.P \mid \bar{a}.Q$$

is represented in the form:
PAR(PREFIX(ACTION(ACNAME "a"), PROCESS "P"), PREFIX(ACTION(coACNAME "a"), PROCESS "Q")), which is a tree structure, see figure 1. There are several shortcuts to construct MR expressions: An infixed backslash (\) between a lambda and a process constructs a PREFIX. Two vertical bars (||) between processes constructs a PAR, and a prefixed bang (!) constructs REP (replication).

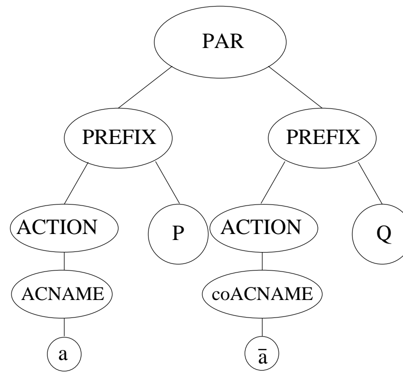

Figure 1: Abstract Syntax Tree

1.2.1 Datatypes for Prefixes

The `pi` datatype is defined by:

$$\pi \quad ::= \quad \tau \mid \lambda \mid \alpha\overline{\delta} \mid \overline{\delta} \rhd \delta \mid (\widetilde{n})\overline{\delta} \rhd\!: p \mid \ \rhd \delta : p \mid \delta \rhd\!: p \mid (\widetilde{n}) \rhd \overline{\delta} : p \tag{6}$$

The constructors that correspond to 6 are: TAU, ACTION, DIRAC/coDIRAC, MOVE/coMOVE, EXIT, ENTER, TAKE and GIVE.

The transition semantics in Table 2 also defines a subset hereof, called β, to distinguish the outgoing actions from as slot. The β datatype is not defined in `MRsim`, a function (called `nest`) is used to recognize the allowable transitions from within a slot.

1.3 Structural Equivalence

E_1	$p \parallel 0 \equiv p$	E_4	$(n)0 \equiv 0$	E_7	$!p \equiv p \parallel !p$
E_2	$p \parallel q \equiv q \parallel p$	E_5	$(n)p \parallel p' \equiv (n)(p \parallel p')$, if $n \notin fn(p')$		
E_3	$(p \parallel p') \parallel p'' \equiv p \parallel (p' \parallel p'')$	E_6	$(n)m\lfloor p \rfloor \equiv m\lfloor (n)p \rfloor$, if $n \neq m$		

Table 1: Structural equivalence.

The rules in Table 1 are implemented as follows: Rule E_1 is handled by the process expression `rmnil` in mr.sml. Note that the function does not neccesarily remove all NIL in a function. Rules E_2 and E_3 are handled by the function `transi`. Rule E_4 is not handled, but does not affect outcomes of transitions. Rule E_5 is handled by the function `scopeext`, which transforms an expression to a normal form. This is done before applying the transition function. Rule E_6 is not handled, it is up to the user to bring an expression to normal form. Rule E_7 is handled in `transi` (case `REP`).

1.4 Transitions

The transition rules are defined in Table 2. The `transi` function takes an expression of type `prex` and returns a (pi * prex) list of possible transitions. The rules $prefix$, $rest$, rep, par, sym, $enter$ and $exit$ are handled directly in `transi`. Rules $openI/II$ are handled in the function `opens`. Rules $co-move$, $give$ and $take$ are handled by the function `ttau`, which is called from `transi`'s PAR case. The $nesting$ rule is handled by `nest`, which recognizes allowable transitions from within slots (β actions). The move/co-move synchronization (rule $sync$) is handled in the function `comp`, as co-moves can be "born" by an exit/enter pair, thus requiring a second scan of the resulting transitions. Table 3, contains the MRsim syntax for transitions, i.e. how the output from the program should be read.

$$(prefix)\ \frac{}{\lambda.p \xrightarrow{\lambda} p} \qquad (rest)\ \frac{p \xrightarrow{\pi} p'}{(n)p \xrightarrow{\pi} (n)p'}\ n \notin fn(\pi) \cup bn(\pi) \qquad (rep)\ \frac{p \parallel !p \xrightarrow{\pi} p'}{!p \xrightarrow{\pi} p'}$$

$$(open\ I)\ \frac{p \xrightarrow{(\tilde{n})\overline{\delta}\triangleright:q} p'}{(n)p \xrightarrow{(n\tilde{n})\overline{\delta}\triangleright:q} p'}\ n \in fn(q)\backslash(fn(\delta) \cup \tilde{n}) \quad (open\ II)\ \frac{p \xrightarrow{(\tilde{n})\triangleright\overline{\delta}:q} p'}{(n)p \xrightarrow{(n\tilde{n})\triangleright\overline{\delta}:q} p'}\ n \in fn(q)\backslash(fn(\delta) \cup \tilde{n})$$

$$(par)\ \frac{p \xrightarrow{\pi} p'}{p \parallel q \xrightarrow{\pi} p' \parallel q} fn(q) \cap bn(\pi) = \emptyset \qquad (sym)\ \frac{p \parallel q \xrightarrow{\pi} p' \parallel q}{q \parallel p \xrightarrow{\pi} q \parallel p'}$$

$$(give)\ \frac{p_1 \xrightarrow{(\tilde{n})\overline{\delta_1}\triangleright:q} p_1' \qquad p_2 \xrightarrow{\delta_1\triangleright\overline{\delta_2}} p_2'}{p_1 \parallel p_2 \xrightarrow{(\tilde{n})\triangleright\overline{\delta_2}:q} (p_1' \parallel p_2')} fn(p_2) \cap \tilde{n} = \emptyset \qquad (take)\ \frac{p_1 \xrightarrow{\triangleright\delta_2:q} p_1' \qquad p_2 \xrightarrow{\delta_1\triangleright\overline{\delta_2}} p_2'}{p_1 \parallel p_2 \xrightarrow{\delta_1\triangleright:q} p_1' \parallel p_2'}$$

$$(sync)\ \frac{p_1 \xrightarrow{(\tilde{n})\overline{\pi}} p_1' \qquad p_2 \xrightarrow{\pi} p_2'}{p_1 \parallel p_2 \xrightarrow{\tau} (\tilde{n})(p_1' \parallel p_2')} fn(p_2) \cap \tilde{n} = \emptyset \qquad (co-move)\ \frac{p_1 \xrightarrow{(\tilde{n})\overline{\delta_1}\triangleright:q} p_1' \qquad p_2 \xrightarrow{\triangleright\delta_2:q} p_2'}{p_1 \parallel p_2 \xrightarrow{\overline{\delta_1}\triangleright\delta_2} (\tilde{n})(p_1' \parallel p_2')} fn(p_2) \cap \tilde{n} = \emptyset$$

$$(nesting)\ \frac{p \xrightarrow{\beta} p'}{n\lfloor p \rfloor \xrightarrow{\beta\cdot n} n\lfloor p' \rfloor}\ n \notin bn(\beta) \qquad (enter)\ \frac{}{n\lfloor \bullet \rfloor \xrightarrow{\triangleright n:p} n\lfloor p \rfloor} \qquad (exit)\ \frac{}{n\lfloor p \rfloor \xrightarrow{\overline{n}\triangleright:p} n\lfloor \bullet \rfloor}$$

Table 2: Transition rules.

$>$ d : P	=	Enter action, direction path d and process P
(n)'d $>$: P	=	Exit action, with restriction n, direction path d and process P
d $>$: P	=	Take action with direction path d and process P
(n) $>$' d : P	=	Give action with restriction n, direction path d and process P
a$-$ $>$ b	=	Move action, from location at address a to location at b
'a$-$ $>$ b	=	coMove action, from location at address a to location at b
tau	=	tau, the silent action

Table 3: Transition syntax in MRsim

1.5 Restrictions

Restrictions are handled by an number of functions:
`cbp`: Compare β and π from a tuple list, and return the list where $\beta = \pi$.
`fln`: Find free names in λ prefixes.
`cn`: Compares a list of binding names with a list of prex's and returns the prex's that do not contain the bound names.
`cln`: Compares two lists. If a member of one list is equal to a member of the other list, return true, else return false.
`cpn`: Compares a list of bound names with free names of π from a `(pi * prex)` tuple list. Only returns tuple if π does not contain bound names from the given list.
`frn`: Finds free names of prexs and actions.

2 Examples

2.1 Alice, Bob, Desk

We apply `MRsim` to the first example given in [1], where the object is to move a resource `C` from Alice to Bob, where Alice and Bob do not know the names of each others' slots.

2.1.1 Move resource from Alice to Bob

The example contains the Alice, Bob, Desk processes, given by:

$$\begin{aligned} Alice &= (a)(a\lfloor C \rfloor \mid Alice') \\ Bob &= (b)(b\lfloor \bullet \rfloor \mid Bob') \\ Desk &= !d\lfloor \bullet \rfloor \\ P &= Alice \mid Bob \mid Desk \end{aligned}$$

The example is provided in the file "examples.sml". *Alice'* is given by

$$Alice' = a \triangleright \overline{d}.0$$

Run the transition on P, by typing

```
- t P;
```

which yields a (pi * prex) tuple list with the possible transitions. To get a more readable format of the list, type

```
- ppal it;
```

which yields (numbered here for convenience):

```
- ppal it;
1) (>d:0, (a.b)(a[C]|a->d.0|b[]|d->b.0|0|!d[]|d[]))
2) (d>:0, (a.b)(0|!d[]|d[]|a[C]|0|b[]|d->b.0))
3) ((a.b)>'d:C, a[]|b[]|d->b.0|!d[])
4) (tau, (a.b)(a[]|0|b[]|0|0|!d[]|d[C]))
> val it = () : unit
```

This produces the number of possible transitions and the resulting expression derived from `P`, given as a 2-tuple list. The first transition (labelled 1 here, for convenience) is an enter action, indicating that the slot `d` is ready to accept a resource. The second transition is a take, derived from a an enter (labelled 1 in the above) and a co-move, which is not visible to the user, because of restrictions on `a` and `b`. The third transition shows the resource `C` exiting it's slot `a`. Because of the *open* rule, this exit action now "carries" the restrictions `a` and `b`. The final transition is a silent action tau, which is the result of synchronization between the move action and a hidden co-move. It is up to the user to decide which transition best expresses what he is trying to model. In this case, it is the tau transition. Note that the restriction notation uses a dot (.) to separate restriction names.
In MRsim, slots should in principle all be restricted toward the environment, to prohibit an arbitrary number of processes from entering the slots, producing `enter` actions, (and `take` if a `move` action is present). If this is not the case, then as illustrated above we only choose to insert the inactive process `NIL`.

In our running example, we now wish to move the resource C to b, i.e. so C is in Bob's possesion, in the slot b. As a practical measure, we set a restriction on the slot `d`, to hide enter actions from the replicated slot `d`. We also hide the exit action from slot d[C], which combined with an enter action would produce a co-move. We do this by definfing:

$$Bob' = d \triangleright \overline{b}.S$$

so we have

$$P' = (a.b.d)(A'||Bob||d \triangleright \overline{b}.S||d\lfloor C \rfloor||!d\lfloor \bullet \rfloor)$$

Running this through MRsim, we get:

```
- ppal (t P');
(tau, (a.b.d)(S|!d[]|d[]|a[]|S|b[C]|S))
> val it = () : unit
```

The resource C has been moved from slot d to slot b. The user only observes the silent action tau, the "details" are hidden by restrictions.

2.1.2 Synchronous Communication

Suppose Bob wants to communicate with the resource C, which resides in the slot b. He does this by using a *directed action*, which is an action that holds information of the the address that the action is supposed to have effect. We define:

$$\begin{aligned} Bob'' &= \overline{c}b.Bob''' \\ C &= c.C' \end{aligned}$$

which gives us:

$$P'' = (a.b.d)(A'||\overline{c}b.Bob'''||b\lfloor c.C'\rfloor||!d\lfloor\bullet\rfloor)$$

The transition function returns:

```
- ppal (t P'');
(tau, (a.b.d)(S|!d[]|a[]|S|b[C']|S))
> val it = () : unit
```

Which shows us that Bob has communicated with the resource C, resulting in a silent action. Had we lifted the restriction d, we would of course have seen the action/co-action in the output. To finish the example, we demonstrate movement from an arbitrarily deep sub-location to an empty slot at another location: If we have

$$\begin{aligned} Bob'' &= c'b \rhd \overline{d}.Bob''' \\ C &= c'\lfloor C'\rfloor \end{aligned}$$

we get the expression:

$$P''' = (a.b.d)(A'||c'b \rhd \overline{d}.Bob'''||b\lfloor c'.\lfloor C'\rfloor\rfloor||!d\lfloor\bullet\rfloor)$$

Reducing this in MRsim gives us:

```
- ppal (t P''');
(tau, (a.b.d)(S|a[]|S|b[c'[]]|S|0|!d[]|d[C']))
> val it = () : unit
```

Resource `C` has moved from the location at `b.c'` to `c`.

2.2 Conclusion and Further Work

We have given a program which makes it possible to do automated transitions on expressions in the Mobile Resource Calculus. There are several possible enhancements, eg. resolving the limitation of closed slots. However, the most interesting enhancements MRsim would be implementing something similar to the Mobility Workbench [3], where it is possible to compare transitions, show bisimulation, equivalence, etc.

References

[1] Godskesen, Hildebrandt and Sassone: *A Calculus of Mobile Resources*, submitted to CONCUR (2002)

[2] Robin Milner: *Communication and Concurrency*, Prentice Hall (1989)

[3] Björn Victor and Faron Moller: *The Mobility Workbench — A Tool for the π-Calculus*, Department of Computer Systems, Uppsala University, Sweden (1994), Also available as Technical Report ECS-LFCS-94-285, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK

A List of Functions and Their Datatypes

This list is not exhaustive, but represents some functions that are either essential for "operating" MRsim, or is a main helper function, though not usually called from the interactive environment.

```
(*Pretty-print a process expression*)
val pp : prex -> string
(*Pretty-print a list of prex's *)
val pl : prex list -> string list
(*Pretty-print (pi*prex) list (= output from transition function) *)
val ppal : (pi * prex) list -> unit
(*Find tau, exit/enter, give/move, take/move *)
val ttau = fn : (pi * prex) list -> (pi * prex) list -> (pi * prex) list
(*Transition function *)
val t = fn : prex -> (pi * prex) list
```

Prex.sig

```
signature Exp =
  sig

    type rname = string
    type addr  = rname list
    type rest  = rname list

    datatype prex =
      PAR of prex * prex         (*PAR parallel (||) constructor   *)
    | PREFIX of lambda * prex    (*prefix 'action' constructor     *)
    | SLOT of rname * res        (*SLOT is a resource (slot)       *)
    | REST of rest * prex        (*Restriction                     *)
    | REP of prex                (*Replication (!)                 *)
    | PROCESS of string          (*PROCESS (atomic) constructor    *)
    | NIL                        (*Empty Constructor               *)
    (*Prefix datatype and constructors                             *)
    (*Syntax:                                                      *)
    (*L ::= a | r                                                  *)
    (*r ::= ad | d>d'                                              *)
    and lambda =
      ACTION  of action            (*ACTION is the action prefix   *)
       | DIRAC of lambda * addr     (*Directed action               *)
       | MOVE  of addr * addr       (*MOVE will be infixed '>'      *)
    (* Resource expressions.                                       *)
    (* Syntax:                                                     *)
    (* r   ::= ¤|p                                                  *)
    and res =
      EMPTY                   (*slots can be empty, ready to accept*)
       | RESOURCE of prex       (*a resource.                      *)
    (*Action datatype. (Action, coaction and tau)                  *)
    (*Syntax:                                                      *)
    (*action ::= a|'a|tau                                          *)
    and action =
      ACNAME of string
       | coACNAME of string
       | TAU

(*Pi datatype - am I failing to find meaningful names? :-)         *)
(*This is a conservative extension of the above datatype           *)
(* pi::=tau|lambda | codirac | comove | exit | enter | give | take *)
datatype pi =
  LAMBDA of lambda
| coDIRAC of lambda * addr
| coMOVE of addr * addr
| EXIT of rest * addr * prex
| GIVE of rest * addr * prex
| ENTER of addr * prex
| TAKE of addr * prex

exception ppcex of string
val prac  : action -> string
val pradd : string list -> string
val prl   : lambda      -> string
val prpi  : pi -> string
val pp    : prex -> string
val pl    : prex list -> string list
val ppa   : (pi * prex) -> string
val ppal  : (pi * prex) list -> unit

val || : prex * prex -> prex
val \  : lambda * prex -> prex
val !  : prex -> prex

  end
```

Prex.sml

```
(*Prex.sml Søren E. Jacobsen 2001-11-19     *)
(*This file describes the environment for the     *)
(*smart card calculus.     *)

structure Prex : Exp =
  struct
    type rname = string
    type addr  = rname list
    type rest  = rname list
```

```
    exception Action
    exception Frn

    datatype prex =
      PAR of prex * prex         (*PAR parallel (||) constructor   *)
    | PREFIX of lambda * prex    (*prefix 'action' constructor     *)
    | SLOT of rname * res        (*SLOT is a resource (slot)       *)
    | REST of rest * prex        (*Restriction                     *)
    | REP of prex                (*Replication (!)                 *)
    | PROCESS of string          (*PROCESS (atomic) constructor    *)
    | NIL                        (*Empty Constructor               *)
    (*Prefix datatype and constructors                             *)
    (*Syntax:                                                      *)
    (*L ::= a | r                                                  *)
    (*r ::= ad | d>d'                                              *)
    and lambda =
      ACTION  of action            (*ACTION is the action prefix   *)
      | DIRAC of lambda * addr     (*Directed action               *)
      | MOVE  of addr * addr       (*MOVE will be infixed '>'      *)
    (* Resource expressions.                                       *)
    (* Syntax:                                                     *)
    (* r   ::= ¤|p                                                  *)
    and res =
      EMPTY                   (*slots can be empty, ready to accept*)
      | RESOURCE of prex       (*a resource.                       *)
    (*Action datatype. (Action, coaction and tau)                  *)
    (*Syntax:                                                      *)
    (*action ::= a|'a|tau                                          *)
    and action =
      ACNAME of string
      | coACNAME of string
      | TAU

(*Pi datatype - am I failing to find meaningful names? :-)         *)
(*This is a conservative extension of the above datatype           *)
(* pi::=tau|lambda | codirac | comove | exit | enter | give | take *)
datatype pi =
  LAMBDA of lambda
| coDIRAC of lambda * addr
| coMOVE of addr * addr
| EXIT of rest * addr * prex
| GIVE of rest * addr * prex
| ENTER of addr * prex
| TAKE of addr * prex

    (* Declare the infix operator "||" that represents             *)
    (* parallel composition                 *)
    val || = PAR

    (* lambda binds to the right, has higher precedence            *)
    (* than parallel                                               *)
    val \ = PREFIX

    (*Let '!' denote replication. Implicitly prefixed.            *)
    val ! = REP

    (*Pretty-print an action *)
    fun prac (ACNAME(s))   = s
      | prac (coACNAME(s)) = "'" ^ s
      | prac TAU        = "tau"

    (*print address            *)
    fun pradd []   = ""
      | pradd (s::[]) = s
      | pradd (s::xs) = s ^ "." ^ pradd xs

    (*print prefix (lambda)                    *)
    fun prl l =
      case l of
ACTION  a   => (prac a)
      | DIRAC (a,l)   => prl a ^ "." ^ pradd l
      | MOVE (s1, s2) => pradd s1 ^ "->" ^ pradd s2

    exception ppcex of string
    (*Pretty-print an MR-expression                  *)
    fun pp prex =
      (case prex of
 PAR    (e1, e2) => (pp e1) ^ "|" ^ (pp e2)
       | PREFIX (l, a)   => prl l ^ "." ^ pp a
       | SLOT (r, e)    =>
```

```
   (case e of
      EMPTY      => r ^ "[]"
    | RESOURCE e => r ^ "[" ^ pp e ^ "]")
       | PROCESS s       => s
       | REST (r,a)      => "(" ^ pradd r ^ ")(" ^ pp a ^ ")"
       | REP p        => "!" ^ pp p
       | NIL        => "0")

    (*Pretty-print pi *)
    fun prpi pi =
      (case pi of
 LAMBDA l       => prl l
       | coDIRAC (a,l)  => prl a ^ ".'" ^ pradd l
       | coMOVE (s1,s2) => "'" ^ pradd s1 ^ "->" ^ pradd s2
       | EXIT (n,a,p)   =>
   if (null n) then
     "'" ^ pradd a ^ ">:" ^ pp p
   else
     "(" ^ pradd n ^ ")'" ^ pradd a ^ ">:" ^ pp p
       | GIVE (n,a,p)   =>
   if (null n) then
     ">'" ^ pradd a ^ ":" ^ pp p
   else
     "(" ^ pradd n ^ ")>'" ^ pradd a ^ ":" ^ pp p
       | ENTER (a,p)    => ">" ^ pradd a ^ ":" ^ pp p
       | TAKE  (a,p)    => pradd a ^ ">:" ^ pp p )

    (*Print a prex list            *)
    fun pl l = map pp l
    (*Pretty-print a (pi, prex) tuple*)
    fun ppa (a,p) = "(" ^ (prpi a) ^ ", " ^ (pp p) ^ ")"

    (*Pretty-print an (pi, prex) tuple list*)
    fun ppal [] = ()
      | ppal (list as (l::ls))  =
      let
val l' = ppa l
      in
print (l' ^ "\n");
ppal ls
      end
  end
```

mr.sml

```
(*File mr.sml *)
(*Søren E. Jacobsen, 2001-10-31*)
(*Updated 2001-11-06*)
(*Transitions for the MR-calculus.*)

use "Prex.sig";
use "Prex.sml";
open Prex;

infix 5 ||;
infixr 6 \;

use "examples.sml";

fun res2prex (RESOURCE(res)) = res
  | res2prex  _ = NIL

(*val cbp : (pi * 'a) list -> (pi * 'a) list *)
(*Compare Beta and Pi from (pi,prex) list, and return the tuple list *)
(*where pi='b *)
fun cbp [] = []
  | cbp l  = let val (pi,p) = hd l
  in
    case pi of
      LAMBDA la   =>
(case la of
   ACTION a       => (pi,p)::(cbp (tl l))
 | DIRAC (lam,add)    => (pi,p)::(cbp (tl l))
 | _       => cbp (tl l))
    | coDIRAC (lam, add)  => (pi,p)::(cbp (tl l))
    | coMOVE(add1, add2)  => (pi,p)::(cbp (tl l))
    | ENTER (add)       => (pi,p)::(cbp (tl l))
    | EXIT (add)       => (pi,p)::(cbp (tl l))
    | _       => cbp (tl l)
```

```
  end

(*Free names in lambda prefixes *)
fun fln (LAMBDA(ACTION(ACNAME(s))))   = s::[]
  | fln (LAMBDA(ACTION(coACNAME(s)))) = s::[]
  | fln (LAMBDA(ACTION(TAU)))         = []
  | fln (LAMBDA(DIRAC(l,a)))          = (fln (LAMBDA(l)))@a
  | fln (LAMBDA(MOVE(a1,a2)))         = a1@a2
  | fln (coDIRAC(l,a))                = (fln (LAMBDA(l)))@a
  | fln (coMOVE(a1,a2))               = a1@a2
  | fln (EXIT(n,a,p))                     = a
  | fln (GIVE(n,a,p))                     = a
  | fln (ENTER(a,p))                    = a
  | fln (TAKE(a,p))                     = a

(*Compare each member of bns (bound names) with each member of l *)
(*and return members of l not equal to bns *)
fun cn bns [] = []
  | cn [] ys  = ys
  | cn bns ys =
  let
    fun eq x y = x = y
  in
    if List.exists (eq (hd ys)) bns
      then cn bns (tl ys)
    else (hd ys)::(cn bns (tl ys))
  end

(* Compare two lists. If a member of one list is equal to a member in *)
(* the other list, return true, else false *)
fun cln xs [] = false
  | cln [] ys = false
  | cln xs ys =
  let fun eq x y = x = y
  in
    List.exists (eq (hd xs)) ys orelse cln (tl xs) ys
  end

(*Compare a name with names in a list. If name exists, return true
fun cpbn bn [] = false
  | cpbn bn xs =
  let fun eq x y = x = y in
    List.exists (eq bn) xs
  end
*)

(*val 'a cpn = fn : string list -> (pi * 'a) list -> (pi * 'a) list  *)
(*Compare a list of bound names (bn) with free names of pi from      *)
(*(pi,prex) tuple list. Only return tuple if pi does not contain     *)
(*bound names from bn.                                               *)
fun cpn [] xs = xs
  | cpn bn [] = []
  | cpn bn xs =
  let
    val (pi,pr) = hd xs
    val fnpi = fln pi
  in if cln bn fnpi
       then cpn bn (tl xs)
     else (pi,pr)::(cpn bn (tl xs))
  end

(*Find free names of processes, actions and actions *)
exception frnex of string
fun frn (PREFIX(a,p))       = (fln (LAMBDA(a)))@(frn p)
  | frn (PAR(p1,p2))        = (frn p1)@(frn p2)
  | frn (SLOT(n,EMPTY))     = n::[]
  | frn (SLOT(n,(RESOURCE(p)))) = n::(frn p)
  | frn (REST(n,(p))) =
    let
      val ps = frn p
    in
      (cn n ps)
    end
  | frn (REP(p))            = frn p
  | frn (PROCESS(p))        = p::[]
  | frn NIL                 = []

(*Compare Actions in an (pi,prex) tuple. Do two *)
(*actions correspond (action/coaction) to each other? *)
fun ca ((LAMBDA(ACTION(ACNAME(ac1))), p),
```

```
(((LAMBDA(ACTION(coACNAME(ac2)))),p2)))  = (ac1 = ac2)
  | ca ((LAMBDA(ACTION(coACNAME(ac1))), p),
((LAMBDA(ACTION(ACNAME(ac2)))),p2))      = (ac1 = ac2)
  | ca ((LAMBDA(DIRAC(l1,a1)),p1),(coDIRAC(l2,a2),p2)) =
  ca ((LAMBDA(l1),p1),(LAMBDA(l2),p2)) andalso a1 = a2
  | ca ((LAMBDA(DIRAC(ACTION(ACNAME(ac1)),ad1)), p),
(((LAMBDA(DIRAC(ACTION(coACNAME(ac2)),ad2))),p2)))  =
  (ac1 = ac2) andalso (ad1 = ad2)
  | ca ((((LAMBDA(DIRAC(ACTION(coACNAME(ac2)),ad2))),p2)),
   (LAMBDA(DIRAC(ACTION(ACNAME(ac1)),ad1)), p))  =
  (ac1 = ac2) andalso (ad1 = ad2)
  | ca ((coDIRAC(l1,a1),p1),
(LAMBDA(DIRAC(l2,a2)),p2)) =
  ca ((LAMBDA(l1),p1),(LAMBDA(l2),p2)) andalso a1 = a2
  | ca (((LAMBDA(MOVE(a1,a2)),p)),
(coMOVE(a3,a4),p2)) = a1 = a3 andalso a2 = a4
  | ca ((coMOVE(a3,a4),p2),
((LAMBDA(MOVE(a1,a2)),p))) = a1 = a3 andalso a2 = a4
  | ca (_,_) = false

(*Is the pi pair a1 and a2 an exit/enter pair?  *)
fun isee (((EXIT(n,a1,p)),p1),
 (ENTER(a2,p2),p3)) = not (cln n (frn p3))
  | isee ((ENTER(a2,p2),p3),
  ((EXIT(n,a1,p)),p1)) = not (cln n (frn p3))
  | isee (_,_)       = false

(*Insert a resource in a context *)
exception insconex of string
fun inscon (cm as coMOVE(a1,a2)) p1 p2 =
  (case p1 of
     SLOT(n,r)    => if n = (hd a2) then SLOT(n,RESOURCE(p2))
     else SLOT(n,r)
   | PAR(pr1,pr2) => PAR((inscon cm pr1 p2),(inscon cm pr2 p2))
   | p => p)
  | inscon l p1 p2 =
 raise insconex (prpi l ^ " is not a coMove action")

exception eecaexception
(*Produce a comove action for an exit/enter pair      *)
fun eeca ((EXIT(n,a1,pr1),p1), (ENTER(a2,pr2),p2)) =
  let
    val cm = coMOVE(a1,a2)
  in
    if n = []
      then (cm,PAR(p1,(inscon cm p2 pr1)))
    else
      (cm,REST(n,PAR(p1,p2)))
  end
  | eeca ((ENTER(a2,pr2),p2), (EXIT(n,a1,pr1),p1)) =
  let val cm = coMOVE(a1,a2)
  in
    if n = []
      then (cm,PAR(p1,(inscon cm p2 pr1)))
    else
    (cm,REST(n,PAR(p1,p2)))
  end
  | eeca (_,_) = raise eecaexception

(*Is the pi pair an exit/move pair? *)
fun isem ((EXIT(n,a1,p),p1),
  ((LAMBDA(MOVE(a2,a3))),p2))  =
  if (a1 = a2) andalso not (cln n (frn p2)) then true else false
  | isem (((LAMBDA(MOVE(a2,a3))),p2),
  (EXIT(n,a1,p),p1)) =
    if (a1 = a2) andalso
      not (cln n (frn p2)) then true else false
  | isem (_,_) = false

(*Produce a give action for an exit/move pair *)
exception eegaex
fun emga ((EXIT(n,a1,p),p1),
 (LAMBDA(MOVE(a2,a3)),p2)) = (GIVE(n,a3,p),PAR(p1,p2))
  | emga ((LAMBDA(MOVE(a2,a3)),p2),
  (EXIT(n,a1,p),p1))       = (GIVE(n,a3,p),PAR(p1,p2))
  | emga (_,_) = raise eegaex

(*Is the pair an enter/move pair? *)
fun isenm ((ENTER(a1,p),p1),((LAMBDA(MOVE(a2,a3))),p2)) = (a1 = a3)
  | isenm ((LAMBDA(MOVE(a2,a3)),p2),(ENTER(a1,p),p1)) = (a1 = a3)
```

```
  | isenm (_,_) = false

(*Produce a take action for an enter/move pair *)
exception emtaex
fun emta ((ENTER(a,p),p1),
 (LAMBDA(MOVE(a1,a2)),p2)) = (TAKE(a2,p),PAR(p1,p2))
  | emta ((LAMBDA(MOVE(a1,a2)),p2),
  (ENTER(a,p),p1))      = (TAKE(a2,p),PAR(p1,p2))
  | emta (_,_) = raise emtaex

(*Handle possible tau/give/take/exit/enter transitions by         *)
(* comparing two lists of (action * prex) tuples                  *)
fun ttau l1 [] = []
  | ttau [] l2 = []
  | ttau l1 l2 =
  let
    val p1 as (a1,pr1) = hd l1
    val p2 as (a2,pr2) = hd l2
  in
    (*case sync ((directed) action/coaction) Move/comove is       *)
    (*handled in a second scan, as the comove actions are "born"  *)
    (*in this first scan.                                          *)
    if ca(p1,p2)
      then ((LAMBDA(ACTION(TAU))),(pr1||pr2))::
(ttau l1 (tl l2))@(ttau (tl l1) l2)
    (*case coMOVE *)
    else if isee(p1,p2)
   then (eeca(p1,p2))::(ttau l1 (tl l2))@(ttau (tl l1) l2)
    (*Case give *)
    else if isem (p1,p2)
   then (emga(p1,p2))::(ttau l1 (tl l2))@(ttau (tl l1) l2)
    (*Case take *)
    else if isenm(p1,p2)
   then (emta(p1,p2))::(ttau l1 (tl l2))@(ttau (tl l1) l2)
 else
   (ttau l1 (tl l2))@(ttau (tl l1) l2)
  end

(*Remove a move action from an expression. Used in comp       *)
fun rml prex =
(case prex of
    cp as (PAR (p1,p2))    => PAR((rml p1),(rml p2))
   | cp as (PREFIX (la,p1)) =>
      (case la of
 m as MOVE(a1,a2) => rml p1
       | _ => cp)
   | cp as (SLOT (n,r))    =>
 (case r of
    EMPTY      => SLOT(n,EMPTY)
  | RESOURCE p => SLOT(n,RESOURCE(rml (res2prex r))))
   | cp as (REST (r,p1))   => REST(r,(rml p1))
   | cp as (REP (p1))      => REP(rml p1)
   | cp as (PROCESS (s))   => cp
   | cp as NIL             => cp)

(*Take a (pi,prex) list and compare the tuples. If two tuples *)
(*contain a move/comove pair, then synchronize *)
fun comp [] = []
  | comp ((pi,prex)::[]) = []
  | comp (prlist as (pi,prex)::prexs)  =
let
  (*Is the tuple pair move/comove? *)
  fun ismc ((LAMBDA(MOVE(a1,a2)),p1),
    (coMOVE(a3,a4),p2)) = a1 = a3 andalso a2 = a4
    | ismc ((coMOVE(a3,a4),p2),
    ((LAMBDA(MOVE(a1,a2)),p1))) = a1 = a3 andalso a2 = a4
    | ismc (_,_) = false
in
  if ismc ((pi,prex),(hd prexs)) then
    (LAMBDA(ACTION(TAU)), (rml prex))::
    (comp ((pi,prex)::(tl prexs)))@(comp prexs)
  else (comp ((pi,prex)::(tl prexs)))@(comp prexs)
end

(*Helper functions that returns expressions in the correct order.   *)
fun par  p1 (pi,p2) = (pi,PAR(p1,p2))
fun rpar p1 (pi,p2) = (pi,PAR(p2,p1))

(*This function takes a (pi, prex) tuple and returns pi * prex with *)
(*the associated address n. It only returns beta values, i.e.       *)
```

```
(*possible transitions from within a slot.                          *)
fun nest n [] = []
| nest n ((pi,prex)::ps) =
  (case pi of
     LAMBDA la =>
       (case la of
  DIRAC (l,add) => (LAMBDA(DIRAC((l), n::add)),
    SLOT(n,RESOURCE(prex)))::(nest n ps)
| MOVE (a1,a2) => nest n ps
| ac as ACTION a =>
    (case a of
       ACNAME a   => (LAMBDA(DIRAC(ac,n::[])),
      SLOT(n,RESOURCE(prex)))::(nest n ps)
      | coACNAME a => ((coDIRAC(ac,n::[])),
      SLOT(n,RESOURCE(prex)))::(nest n ps)
      | TAU        => ((LAMBDA(ACTION(TAU))),
      SLOT(n,RESOURCE(prex)))::(nest n ps)))
   | coDIRAC (lam, add) => ((coDIRAC(lam, n::add)),
    SLOT(n,RESOURCE(prex)))::(nest n ps)
   | coMOVE (a1,a2)     => ((coMOVE(n::a1, n::a2)),
    SLOT(n,RESOURCE(prex)))::(nest n ps)
   | ENTER (a,p)    => (ENTER(n::a, p),
SLOT(n, RESOURCE(prex)))::(nest n ps)
   | EXIT (m,a,p)   => (EXIT(m, n::a, p),
SLOT(n, RESOURCE(prex)))::(nest n ps)
   | TAKE (a,p)     => nest n ps
   | GIVE (m,a,p)   => nest n ps)

(*val opens = fn : (pi * prex) list -> (pi * prex) list             *)
(*This function handles open I/II of the transition rules. The      *)
(*restricted names should not be the same as free names of the      *)
(*EXIT/ENTER constructor.                                           *)
fun opens [] = []
  | opens ((pi,REST(rest,prex))::ps)  =
  (case pi of
     EXIT (n,a,p) => ((EXIT(rest@n,a,p)),prex)::(opens ps)
   | GIVE (n,a,p) => ((GIVE(rest@n,a,p)),prex)::(opens ps)
   | _              => (pi,REST(rest,prex))::(opens ps))
  | opens ((pi,prex)::ps) =
     (case pi of
EXIT (n,a,p) => ((EXIT(n,a,p)),prex)::(opens ps)
      | GIVE (n,a,p) => ((GIVE(n,a,p)),prex)::(opens ps)
      | _              => (pi,prex)::(opens ps))

(*Remove redundant expressions from a composite expression *)
fun rmprex p prex =
  (case prex of
    cp as (PAR (p1,p2))    => if p = cp then NIL
      else PAR((rmprex p p1),(rmprex p p2))
   | cp as (PREFIX (l,p1)) => if p = cp then NIL
      else if p = p1 then rmprex p p1
   else cp
   | cp as (SLOT (n,r))    => if p = cp then NIL
      else if p = (res2prex r) then
(SLOT(n,RESOURCE((rmprex p (res2prex r)))))
   else cp
   | cp as (REST (r,p1))   => if p = cp then NIL
      else if p = p1 then (REST(r,(rmprex p p1)))
   else cp
   | cp as (REP (p1))      => cp
   | cp as (PROCESS (s))   => if p = cp then NIL else cp
   | cp as NIL             => NIL)

(*Remove NIL exp. from prex. Rule E1 of structural equivalence.*)
fun rmnil prex =
  let
    fun rmn prex =
      (case prex of
 cp as (PAR(NIL,NIL)) => NIL
       | cp as (PAR (p1,NIL)) => rmn p1
       | cp as (PAR(NIL,p1))  => rmn p1
       | cp as (PAR(p1,p2))   =>
   let
     val pr1 = rmn p1
     val pr2 = rmn p2
   in
     PAR(pr1,pr2)
   end
       | _                    => prex)
  in
```

```
    (rmn prex)
  end

(*Find the replications in an expression and return them.     *)
(*This is used when "cleaning up". The redundant replicated   *)
(*processes are later removed (fun clean)                      *)
fun findrep p =
  (case p of
     PAR(p1,p2)  => (findrep p1)@(findrep p2)
   | REST(n,p)   => (findrep p)
   | REP(p)      => p::[]
   | SLOT(n,r)   => findrep (res2prex r)
   | PREFIX(n,p) => findrep p
   | PROCESS p   => []
   | NIL         => [])

(*Find the restrictions in an expression and return them.     *)
(*This is used when "cleaning up". The redundant replicated   *)
(*processes are later removed (fun clean)                      *)
fun findrest p =
  (case p of
     PAR(p1,p2)  => (findrest p1)@(findrest p2)
   | REST(n,p)   => REST(n,p)::[]
   | REP(p)      => findrest p
   | SLOT(n,r)   => findrest (res2prex r)
   | PREFIX(n,p) => findrest p
   | PROCESS p   => []
   | NIL         => [])

(*Remove duplicate prexes (given in a list) from a prex *)
fun rmpl pl prex =
    if not (null pl) then
      rmpl (tl pl) (rmprex (hd pl) prex)
    else prex

(*Remoce NIL and duplicates from (pi * prex) expressions *)
fun rmpil [] = []
  | rmpil ((pi,p)::[]) = (pi, (rmnil (rmpl (findrep p) p)))::[]
  | rmpil ((pi,p)::prexs) = (pi,(rmnil (rmpl (findrep p) p)))::(rmpil prexs)

(*Clean a (pi * prex) list by removing duplicate transitions  *)
fun cltrans [] = []
  | cltrans ((pi,prex)::[])  = (pi,prex)::[]
  | cltrans ((pi1,p1)::prexs) =
  let
    val pr = tl prexs
    fun neq a1 a2 = not (a1 = a2)
  in
    (pi1,p1)::(cltrans (List.filter (neq (pi1,p1)) prexs))
  end

(*Scope extension. Rule E5 in rules for structural equivalence      *)
exception scopeex of string
fun scopeext (prex as PAR(REST(n1,p1),REST(n2,p2))) =
  if not (cln n1 (frn p2)) andalso not (cln n2 (frn p1)) then
    REST(n1@n2,PAR(p1,p2))
  else prex
   | scopeext (prex as PAR(REST(n,p1), p2)) =
    let
      val pr1 = scopeext p1
      val pr2 = scopeext p2
    in
      if not (cln n (frn p2)) then
(REST(n,PAR(pr1, pr2)))
      else prex
    end
  | scopeext (prex as PAR(p2,REST(n,p1)))  =
    let
      val pr1 = scopeext p1
      val pr2 = scopeext p2
    in
      if not (cln n (frn p2)) then
(REST(n,PAR(pr1,pr2)))
      else prex
    end
  | scopeext (prex as (PAR(p1,p2))) =
    let
      val pr1 = scopeext p1
      val pr2 = scopeext p2
      val pr  = PAR(pr1,pr2)
```

```
    in
      if  pr = prex then
pr
      else (scopeext pr)
    end
  | scopeext p = p

exception transiexception of string
(*This transition function examines possible transitions in a       *)
(*prex and returns a (pi * prex) list     *)
(*case par.  *)
fun transi (PAR(p1,p2)) =
  let
    val pr1  = transi p1
    val pr2  = transi p2
    val l    = (ttau pr1 pr2)@(map (par p1) pr2)@
      (map (rpar p2) pr1)
    val l2   = cltrans (comp l)
  in
    l@l2
  end
  (*case prefix*)
  | transi (PREFIX(l,r)) =
  (case l of
     ACTION a     => (LAMBDA(ACTION(a)),r)::[]
   | DIRAC (l,a)  => (LAMBDA(DIRAC(l,a)),r)::[]
   | MOVE (a1,a2) => (LAMBDA(MOVE(a1,a2)),r)::[])
  (*case rest*)
  | transi (REST(r,p)) =
     let
       fun rest n (pi,pr) = (pi,REST(n,pr))
       val ps  = transi p
       val psr = map (rest r) ps
     in
       opens (cpn r psr)
     end
  (*case rep*)
  | transi (REP p) =
     let
       (*rmprex and rmnil *)
       fun clean l (pi,prex) = (pi,(rmnil (rmprex l prex)))
       (*clean a (pi,prex) list *)
       fun cleanpl l prexs = cltrans (map (clean l) prexs)
       val l  = transi (PAR(PAR(p,p),p))
       val cl = cleanpl p l
     in
       if l<>[] then map (rpar (PAR((REP p),p))) cl
       else []
     end
  (*case nesting, enter, exit *)
  | transi (s as SLOT(n,r)) =
     (case r of
EMPTY => (ENTER(n::[],NIL),SLOT(n,EMPTY))::[]
      | RESOURCE p => let
  val l  = transi p
  val l' = cbp l
in
  ((EXIT([],n::[],(res2prex(r)))),
   (SLOT(n,EMPTY)))::(nest n l')
end)
  | transi (PROCESS p) = []
  | transi (NIL)       = []

(*Putting it all together *)
fun t p = cltrans (rmpil (transi (scopeext p)))

(*Same as above, with infixed composition *)
fun tr p = (((cltrans o rmpil) o transi) o scopeext) p

(*A counter. Not used.*)
fun count n =
  let
    val p = 1
    fun ct n m =
      if (n >= m) then
(print ((Int.toString m) ^ "\n"); (ct n (m+1)))
      else print "";
  in
    ct n p
  end
```