

# Introduction to mCRL2 (modelling)

Luís S. Barbosa

HASLab - INESC TEC  
Universidade do Minho  
Braga, Portugal

15 April, 2015

# mCRL2: A toolset for process algebra

mCRL2 provides:

- a generic **process algebra**, based on ACP (Bergstra & Klop, 82), in which other calculi can be **embedded**
- extended with **data** and (real) **time**
- with an **axiomatic** semantics
- the full  **$\mu$ -calculus** as a specification logic
- powerful toolset for **simulation** and **verification** of reactive systems

[www.mcrl2.org](http://www.mcrl2.org)

# Actions

## Interaction through multisets of actions

- A **multiaction** is an elementary unit of interaction that can **execute itself atomically in time** (no duration), after which it terminates successfully

$$\alpha ::= \tau \mid a \mid a(d) \mid \alpha \mid \alpha$$

- actions may be parametric on **data**
- the structure  $\langle N, |, \tau \rangle$  forms an Abelian **monoid**

# Sequential processes

## Sequential, non deterministic behaviour

The set  $\mathbb{P}$  of **processes** is the set of all terms generated by the following BNF, for  $a \in N$ ,

$$p ::= \alpha \mid \delta \mid p + p \mid p \cdot p \mid P(d)$$

- **atomic process**:  $a$  for all  $a \in N$
- **choice**:  $+$
- **sequential composition**:  $\cdot$
- **inaction or deadlock**:  $\delta$  (it cannot even to terminate!)
- **process references** introduced through definitions of the form  $P(x : D) = p$ , parametric on **data**

# Sequential Processes

## Exercise

Describe the behaviour of

- $a.b.\delta.c + a$
- $(a + b).\delta.c$
- $(a + b).e + \delta.c$
- $a + (\delta + a)$
- $a.(b + c).d.(b + c)$

## Axioms: $+$ , $\cdot$ , $\delta$

$$\begin{array}{lll} A1 & x + y & = y + x \\ A2 & (x + y) + z & = x + (y + z) \\ A3 & x + x & = x \\ A4 & (x + y).z & = x.z + y.z \\ A5 & (x.y).z & = x.(y.z) \\ A6 & x + \delta & = x \\ A7 & \delta \cdot x & = \delta \end{array}$$

- the equality relation is **sound**: if  $s = t$  holds for basic process terms, then  $s \sim t$
- and **complete**: if  $s \sim t$  holds for basic process terms, then  $s = t$
- an axiomatic theory enables **equational reasoning**

## Axioms: $+$ , $\cdot$ , $\delta$

$$\begin{array}{lll} A1 & x + y & = y + x \\ A2 & (x + y) + z & = x + (y + z) \\ A3 & x + x & = x \\ A4 & (x + y).z & = x.z + y.z \\ A5 & (x.y).z & = x.(y.z) \\ A6 & x + \delta & = x \\ A7 & \delta \cdot x & = \delta \end{array}$$

- the equality relation is **sound**: if  $s = t$  holds for basic process terms, then  $s \sim t$
- and **complete**: if  $s \sim t$  holds for basic process terms, then  $s = t$
- an axiomatic theory enables **equational reasoning**

# Axioms: $\delta$ , $+$ , $\cdot$ , $\delta$

## Exercise

- show that  $\delta.(a + b) = \delta \cdot a + \delta \cdot b$
- show that  $a + (\delta + a) = a$
- is it true that  $a.(b + c) = a.b + a.c$  ?



# mCRL2: A toolset for process algebra

## Example

```
act    order, receive, keep, refund, return;

proc   Buy = order.OrderedItem

      OrderedItem = receive.ReceivedItem + refund.Buy;
      ReceivedItem = return.OrderedItem + keep;

init   Buy;
```

# Example

## Clock

```
act    set, alarm, reset;  
  
proc  P = set.R  
      R = reset.P + alarm.R  
  
init  P
```

# Example

## A refined clock

```
act    set:N, alarm, reset, tick;

proc   P = (sum n:N . set(n).R(n)) + tick.P
       R(n:N) = reset.P + ((n == 0) -> alarm.R(0) <> tick.R(n-1))

init  P
```

# Parallel composition

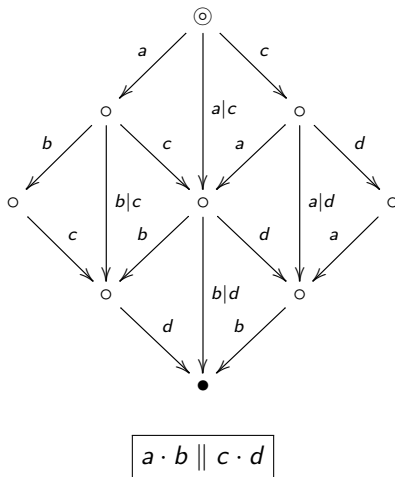
$\parallel$  = interleaving + synchronization

- modelling principle: interaction is the key element in software design
- modelling principle: (distributed, reactive) architectures are configurations of communicating black boxes
- mCRL2: supports flexible synchronization discipline ( $\neq$  CCS)

$$p ::= \dots \mid p \parallel p \mid p \mid p \mid p \parallel p$$

# Parallel composition

## An example



# Parallel composition

- **parallel**  $p \parallel q$ : interleaves and synchronises the actions of both processes.
- **synchronisation**  $p | q$ : synchronises the first actions of  $p$  and  $q$  and combines the remainder of  $p$  with  $q$  with  $\parallel$ , cf axiom:

$$(a.p) | (b.q) \sim (a | b).(p \parallel q)$$

- **left merge**  $p \ll q$ : executes a first action of  $p$  and thereafter combines the remainder of  $p$  with  $q$  with  $\parallel$ .

# Parallel composition

## A semantic parenthesis

**Lemma:** There is no sound and complete finite axiomatisation for this process algebra with  $\parallel$  modulo bisimilarity [F. Moller, 1990].

**Solution:** combine two auxiliary operators:

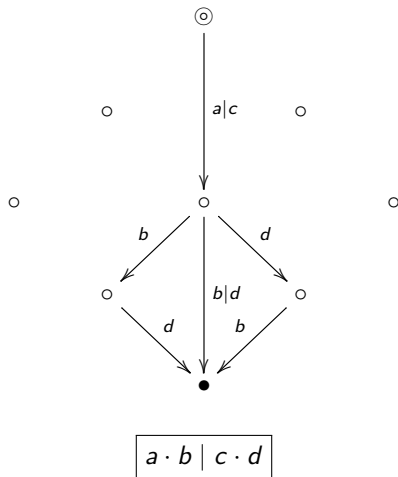
- left merge:  $\ll$
- synchronous product:  $|$

such that

$$p \parallel t \sim (p \ll t + t \ll p) + p | t$$

# Parallel composition

## An example





# Interaction

## Communication $\Gamma_C(p)$ (com)

- applies a **communication function**  $C$  forcing action synchronization and renaming to a new action:

$$a_1 \mid \cdots \mid a_n \rightarrow c$$

- data parameters are retained in action  $c$ , e.g.

$$\Gamma_{\{a|b \rightarrow c\}}(a(8) \mid b(8)) = c(8)$$

$$\Gamma_{\{a|b \rightarrow c\}}(a(12) \mid b(8)) = a(12) \mid b(8)$$

$$\Gamma_{\{a|b \rightarrow c\}}(a(8) \mid a(12) \mid b(8)) = a(12) \mid c(8)$$

- left hand-sides in  $C$  must be disjoint: e.g.,  $\{a \mid b \rightarrow c, a \mid d \rightarrow j\}$  is not allowed

# Interface control

Restriction:  $\nabla_B(p)$  (**allow**)

- specifies which actions are allowed to occur
- disregards the data parameters of actions

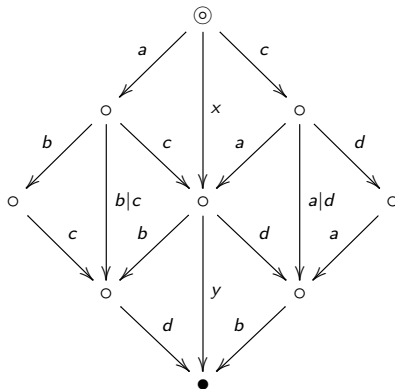
$$\nabla_{\{d,b|c\}}(d(12) + a(8) + (b(\text{false}, 4) | c)) = d(12) + (b(\text{false}, 4) | c)$$

- $\tau$  is always allowed to occur

Discuss:  $\nabla_{\{x,y\}}(\Gamma_{\{a|c \rightarrow x, b|d \rightarrow y\}}(a.b \parallel c.d))$

# Interface control

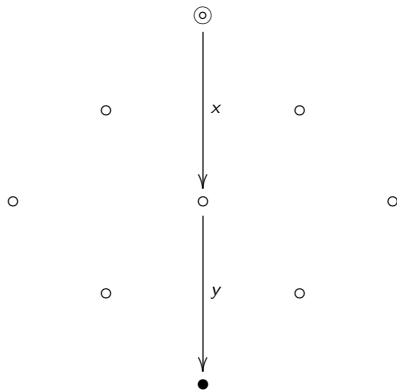
## An example



$$\Gamma_{\{a|c \rightarrow x, b|d \rightarrow y\}}(a.b \parallel c.d)$$

# Interface control

## An example



$$\nabla_{\{x,y\}}(\Gamma_{\{a|c \rightarrow x, b|d \rightarrow y\}}(a.b \parallel c.d))$$

# Interface control

Block:  $\partial_B(p)$  (**block**)

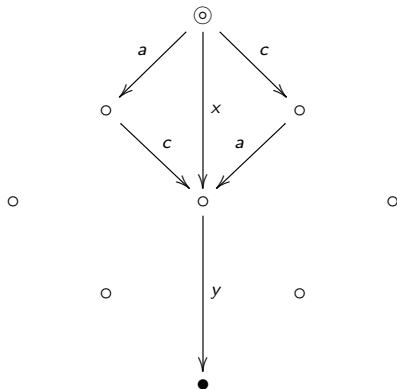
- specifies which actions are not allowed to occur
- disregards the data parameters of actions

$$\partial_{\{b\}}(d(12) + a(8) + (b(\text{false}, 4) \mid c)) = d(12) + a(8)$$

- the effect is that of renaming to  $\delta$
- $\tau$  cannot be blocked

# Interface control

## An example



$$\partial_{\{b,d\}}(\Gamma_{\{b|d \rightarrow y\}}(a.b \parallel c.d))$$

# Interface control

## Enforce communication

- $\nabla_{\{c\}}(\Gamma_{\{a|b \rightarrow c\}}(p))$
- $\partial_{\{a,b\}}(\Gamma_{\{a|b \rightarrow c\}}(p))$

# Interface control

## Renaming $\rho_M(p)$ (rename)

- renames actions in  $p$  according to a mapping  $M$
- also disregards the data parameters, but when a renaming is applied the values of data parameters are retained:

$$\begin{aligned}\rho_{\{d \rightarrow h\}}(d(12) + s(8) \mid d(\text{false}) + d.a.d(7)) \\ = h(12) + s(8) \mid h(\text{false}) + h.a.h(7)\end{aligned}$$

- $\tau$  cannot be renamed



# Interface control

## Hiding $\tau_H(p)$ (**hide**)

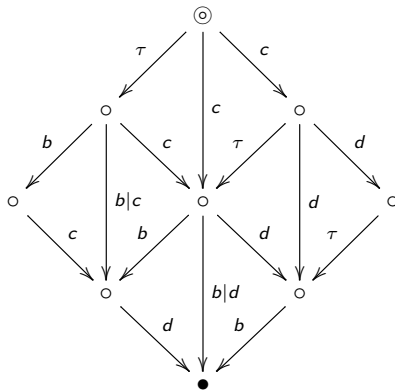
- hides (or renames to  $\tau$ ) all actions in  $H$  in all multiactions of  $p$ .
- disregards the data parameters

$$\begin{aligned} \tau_{\{d\}}(d(12) + s(8) \mid d(\text{false}) + h.a.d(7)) \\ = \tau + s(8) \mid \tau + h.a.\tau = \tau + s(8) + h.a.\tau \end{aligned}$$

- $\tau$  and  $\delta$  cannot be renamed

# Interface control

## An example



$$\tau_{\{a\}}(\Gamma_{\{b|d \rightarrow y\}}(a.b \parallel c.d))$$

# Example

## New buffers from old

```
act   inn, outt, ia, ib, oa, ob, c : Bool;

proc  BufferS = sum n: Bool.inn(n).outt(n).BufferS;

      BufferA = rename({inn -> ia, outt -> oa}, BufferS);
      BufferB = rename({inn -> ib, outt -> ob}, BufferS);

      S = allow({ia, ob, c}, comm({oa|ib -> c}, BufferA || BufferB));

init  hide({c}, S);
```

# Data types

- **Equalities:** equality, inequality, conditional ( $\text{if}(-,-,-)$ )
- **Basic types:** booleans, naturals, reals, integers, ... with the usual operators
- **Sets, multisets, sequences** ... with the usual operators
- **Function definition**, including the  $\lambda$ -notation
- **Inductive types:** as in

```
sort   BTree = struct leaf(Pos) | node(BTree, BTree)
```

# Signatures and definitions

Sorts, functions, constants, variables ...

```
sort  S, A;
```

```
cons  s,t:S, b:set(A);
```

```
map   f:  S x S -> A;  
      c:  A;
```

```
var   x:S;
```

```
eqn   f(x,s) = s;
```

# Signatures and definitions

A full functional language ...

```
sort   BTree = struct leaf(Pos) | node(BTree, BTree);

map    flatten: BTree -> List(Pos);

var    n:Pos, t,r:BTree;

eqn    flatten(leaf(n)) = [n];
        flatten(node(t,r)) = flatten(t) ++ flatten(r);
```

# Processes with data

## Why?

- Precise modeling of real-life systems
- Data allows for finite specifications of infinite systems

## How?

- data and processes parametrized
- summation over data types:  $\sum_{n:N} s(n)$
- processes conditional on data:  $b \rightarrow p \diamond q$

# Examples

## A counter

```
act    up, down;
       setcounter:Pos;

proc   Ctr(x:Pos) = up.Ctr(x+1)
       + (x>0) -> down.Ctr(x-1)
       + sum m:Pos.(setcounter(m).Ctr(m))

init   Ctr(345);
```



# Examples

## A dynamic binary tree

```
act    left,right;

map    N:Pos;

eqn    N = 512;

proc   X(n:Pos)=(n<=N)->(left.X(2*n)+right.X(2*n+1))<>delta;

init   X(1);
```